A Conflict-Free Replicated Abstract Syntax Tree

Aäron Munsters
Vrije Universiteit Brussel
Brussels, Belgium
aaron.sarah-louise.munsters@vub.be

Abstract

With the rise of remote work, remote real-time collaborative code editing is gaining popularity while lacking dedicated tooling support. This work tackles this limitation by proposing a technique to provide real-time high-availability code editors through modelling the editor-underlying abstract syntax tree as a conflict-free replicated data-structure. On top of the CRDT implementation, a demo web-based editor for a small Scheme-like language is built to assess the viability of this approach.

1 Introduction

With telecommuting on the rise, collaboratively working on a single code base in real-time is gaining traction. The code editor Visual Studio Code, which is the current leading choice of IDE[1], promotes the Live Share extension as part of its top 50 most installed extensions[2], demonstrating the relevance of providing developers a structured basis to develop in a collaborated manner.

Traditional collaborating tools such as Live Share for Visual Studio Code or Teletype for Atom in essence enable collaboration by representing the code files that are being worked on as large strings which are concurrently manipulated by different replica's, iethe machines on which remote developers work. This technique can be considered a simplified approach to enabling the replication of the code, as code usually takes a strict structural form, which is then completely overlooked by this technique. This brings with it different limitations.

- **Easy to break the code structure**
  When developers write code alone, they tend to make syntactic mistakes. Even though these mistakes are pointed at by the parser, it is not always an easy task to locate the culprit of this case, for example identifying the location at which a matching bracket is missing to correct the program both syntactically and semantically can get complex for code with lots of nested statements. When it is up to two developers to locate the missing matching bracket, parallel development only complicates matters to identify who is to blame and what part of the code to repair for such breaking changes.
- **Need for strong synchronization**
  Traditional collaboration tools enable collaboration by assigning one developer as the host and inviting others to join their development sessions[3] which requires the host to stay live at all times. The host as such could be considered a single point of failure, which may break the application availability upon the occurrence of a network partition.
- **Weak merge strategy:** In case the collaboration tool opts for availability by enabling the developer to continue on a local copy of the code during a network partition, upon a connection reestablishment the tool is uninformed about the code and its structure on how it should be merged.
  It could be considered an easy task to merge little edits, say the addition of a single character. Concurrent changes to the overall program structure, however, such as statement moves and updates can significantly add to the complexity of such merges. Figure 1 illustrates this with a concrete example. The initial code contains two statements, an `eat` and a `touch` statement. For a given network partition, the local edit of changing the `"apple"` string into the string `"banana"` should take place once the connection is reestablished. In case a different remote replica changes the order of the statements, should result in both the update and the move operation having taken place. This complexity of merging both operations increases with the size of the code that is being worked on such as tracking the moved statement across more than neighbouring lines or even in different scopes.

To overcome these limitations this work proposes a novel strategy to provide developers with an editor that models the takes underlying AST that is being edited concurrently through modelling the tree as a conflict-free replicated tree. The source code for the project is available on https://github.com/aaronmunsters/AST_CRDT while the working prototype is live available on https://aaronmunsters.github.io/AST_CRDT/. This report is structured to motivate the new approach in Section 2 while covering technical challenges in Sections 4 and 3. As a small demonstration application is included with this report, as it is discussed in Section 5. Section 6 discusses the set of tests included in the codebase to assert the correct

[1]As is reported by the 2021 Stack Overflow Developer Survey
[2]As was displayed on the Visual Studio Marketplace most popular extensions as of 7 January 2022

[3]Live Share calls these *sessions*, Teletype calls these *portals*.

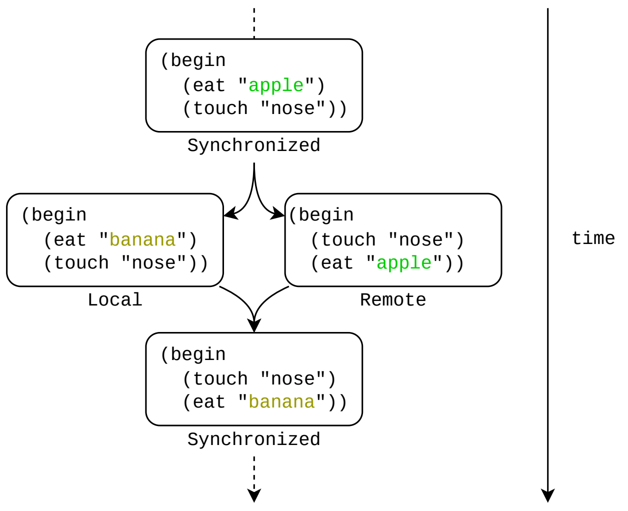

Figure 1. A sample Scheme program that evolves through different machines that temporarily break synchronization. This example shows that reasoning over the code structure is required to successfully merge the two separate operations.

behaviour of the code, while Section 7 covers limitations known to the work and proposes improvements in the form of future work.

2 A Replicated Conflict Free Abstract Syntax Tree

This work proposes a new approach to collaborative code editing through modelling the underlying abstract syntax tree (AST) as a conflict-free replicated datatype [5] (CRDT). The prototype accompanied by this report is an implementation for the Scheme language.

Figure 2 provides an overview of the flow that a single local edit makes through the different stages, each associated with a concrete component, before it enables all replica's to converge to an identical AST. The approach consists of the code editor continuously monitoring (in an event-based manner) for changes taking place on the local replica. For incoming changes in the editor, the altered source code is parsed and the newly constructed AST is mapped onto the previous AST, after which the delta for both ASTs can be computed in the form of an edit script, meaning a script that contains the changes to go from the source AST to the destination AST. This edit script containing the changes is then propagated to all other replica's accompanied by a logical timestamp, forming the basis of the AST as an operation-based CRDT. For the replica's, each incoming edit script is added to its local list of operations, which can be reordered due to the inclusion of the logical timestamp, allowing for an ordering of the events. The ordered list of edits allows the local replica to compute the final resulting AST, which is then reflected in the code editor.

The project is developed in the Scala language, built using structures and libraries that are ScalaJS compliant. This comes with the benefits of the scala type checker asserting type safety while remaining portable in terms of the underlying execution engine, which can be either the JVM or a JavaScript engine.

Approach motivation

Initial work was put into remaining language agnostic, where the AST changes could be detected by an incremental parser such as Escaya for JavaScript or ts-morph for TypeScript such that the reported changes would be propagated to other replica's.
This strategy was rejected as these incremental parsers are built with a single local in-memory AST in mind, where the increments are reported in terms of pointers to AST nodes that have been affected, which would require in-depth adjustments to providing each node with unique identifiers and changing the incremental parsers to report serializable changes that can be shared across the network to devices with isolated memory.
A different strategy thus was opted for to implement a small parser for a Scheme-like language, allowing to remain in full control of the internal representations, opting for a serializable-first design for the AST and the detected changes.

The design of the AST CRDT is highly inspired by the design of the paper "A highly-available move operation for replicated trees" by Kleppmann et al. [3]. Their work discusses the design of a tree CRDT, motivating why move operations become complex for replicated trees. We summarize the discussion on the difficulties:

- **Concurrently moving the same node**
  Say concurrently a move operation takes place for the same node, resulting in the node *green* becoming the child of its sibling *blue* on one replica and becoming the child of its sibling *orange* on another replica as Figure 3 presents. Once these operations merge, different operations are possible, prioritize a single outcome (Figure 3 *(a)* and *(b)*), or duplicate the *green* node so that both *blue* and *orange* can have an instance of *green* as their child (Figure 3 *(c)*). Another option could be to have both nodes *blue* and *orange* have as their child node *green*, however this would break the tree-like structure (Figure 3 *(d)*).
- **Introducing cycles**
  While move operations may uphold the guarantees of the tree remaining as an acyclic structure on individual

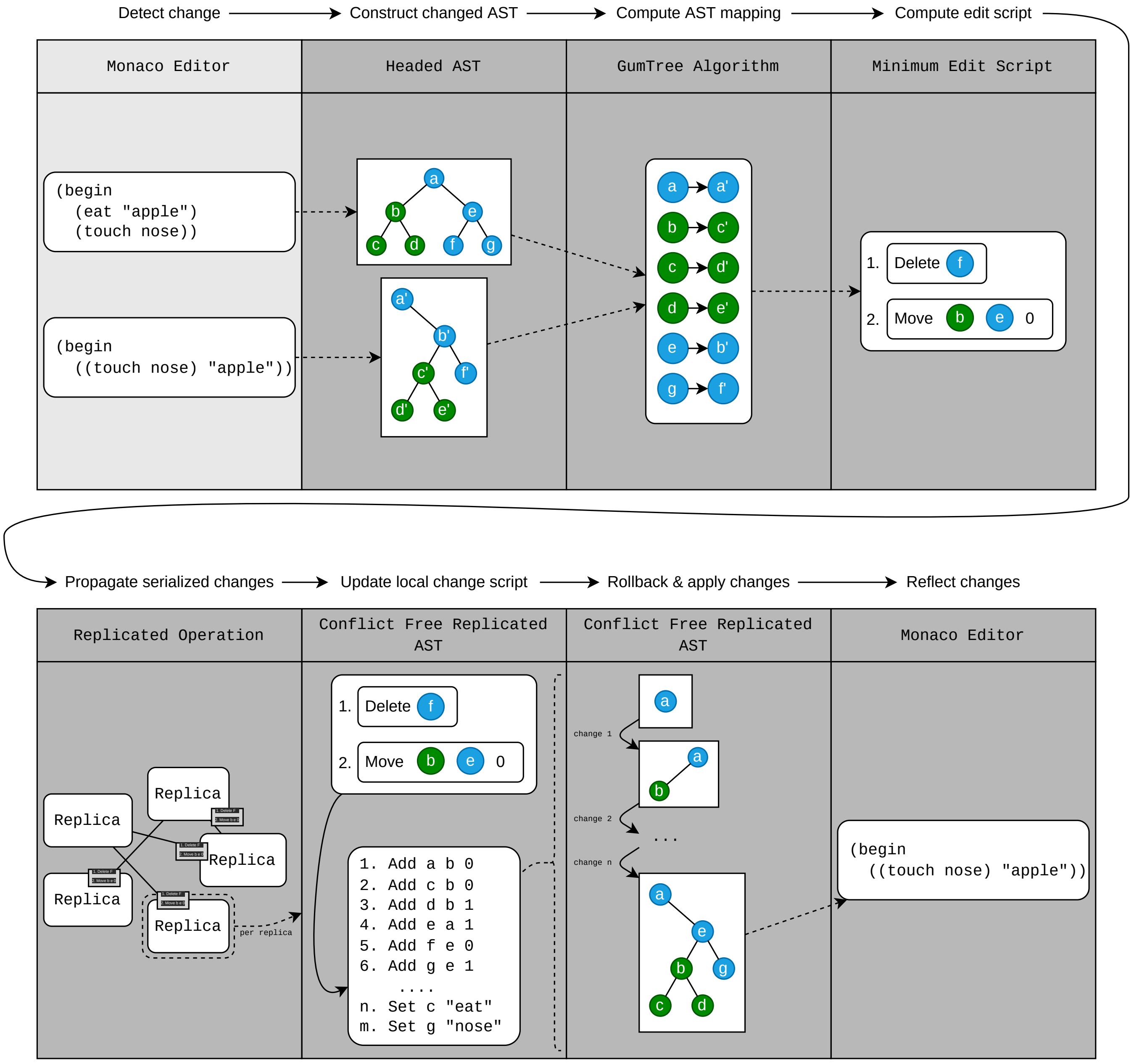

Figure 2. An overview of the stages that flow through different components working together to make up a distributed conflict-free replicated abstract syntax tree. Changes detected by the Monaco Editor trigger the construction of a new HeadedAST, which is then mapped onto the previous HeadedAST through the GumTreeAlgorithm allowing to compute the changes that took place as a MinimumEditScript. These AST changes are serialized and propagated over the wire as ReplicatedOperations, as the CRDT is an operation-based CRDT. The incoming operations allow each replica to roll back the local AST to then reapply all known changes in order. The final computed AST is displayed by the Monaco Editor which then reflects the converging AST to the developer.

replica's, their combination may introduce cycles. Figure 4 shows the distinct outcomes possible. The figure illustrates the possible choices when two sibling nodes are concurrently made a parent of one another. It allows for either move operation to remain (Figure 4 *(a)* and *(b)*), a duplicate instance of both nodes to enable both move operations (Figure 4 *(c)*) or to introduce a cycle to enable both operations without duplication, but this would break the tree requirement (Figure 4 *(d)*).

Their work then proposes to accompany each operation with a timestamp such that incoming operations that are out of order, allowing the CRDT to determine the point at which these should have been applied. It then performs a rollback using the *undo_op* operation up until the point where the incoming operation can be applied after which all latter operations can be replayed using the *redo_op* operation. These are the conditions by which the authors prove convergence in a strong eventual consistent setting.

3 Computing AST Changes

Before propagating AST changes over the network, they need to be determined locally. Computing source-code changes can be performed by incremental compilers, they could be derived from projectional editors[4], or they can be computed by AST tree differencing algorithms[5].

Whereas incremental compilers and projectional editors come with the disadvantage that they are specialized per language, AST tree differencing algorithms exist with *language agnostic* specifications. This project has adopted the GumTree algorithm [2], a language-agnostic algorithm to compute the changes between two ASTs.

Computing changes in trees produces an *edit script*, which is an ordered set of edit operations that when applied to the source tree results in an output of the destination tree. The operations that suffice for tree transformations are captured by the following operations [1]:

- *update(n, v)* to update the content of node *n* with the value of *v*.
- *add(t, p, i, l, v)* to add the node *t* with the label *l* and the value *v* to the parent *p* at index *i*.
- *delete(t)* to delete node *t*.
- *move(t, p, i)* to move node *t* in the tree to be a child of *p* at index *i*.

[4]These are editors that let the user develop their programs through manipulating an AST directly rather than having the programmer type the program in an open space for characters. The benefit of these editors is that the programmer can make no syntactic mistakes, the drawback is that the amount of different building blocks to construct a program can become complex for languages with a rich syntax. An example projectional editor that is actively maintained for the TypeScript language is Forest.

[5]For the reader, a curated list of papers and tools is available on the page "Pointers on abstract syntax tree differencing algorithms and tools".

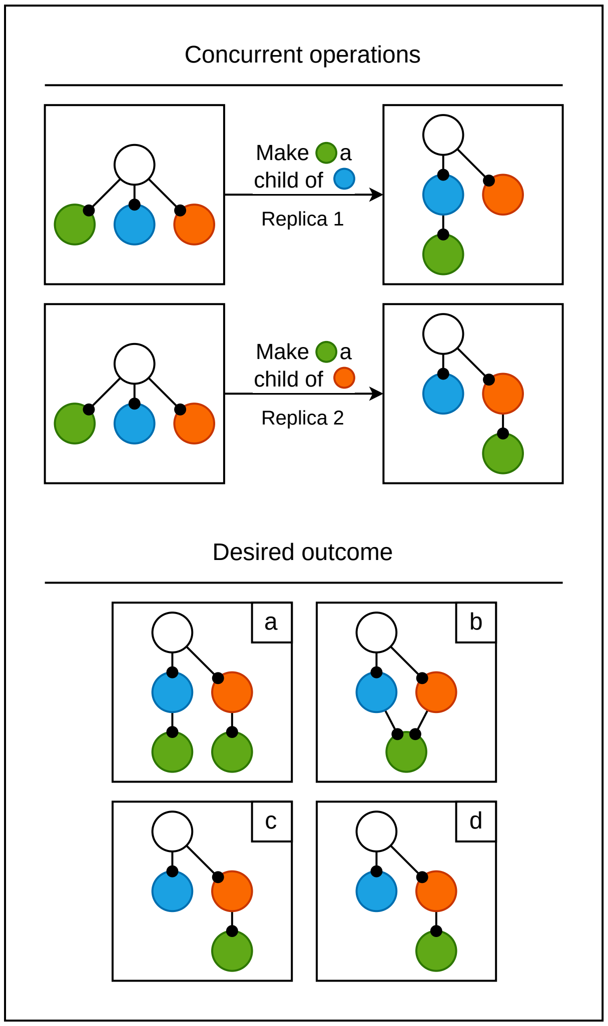

Figure 3. A case in which concurrent move operations take place for a node to become the child of either one of its siblings, including the possible outcomes. The shown outcomes illustrate there is either an operation-discarding choice, a duplicating choice or a tree-structure breaking choice that must be made. Inspired by the illustrations designed by Kleppmann et al. [3].

When spoken of *minimum edit script*, the edit script is the smallest possible ordering of edit operations that is still a valid edit script. In the context of the CRDT AST, the need for remaining as close as possible to a minimum edit script is desirable since it can affect the network load.

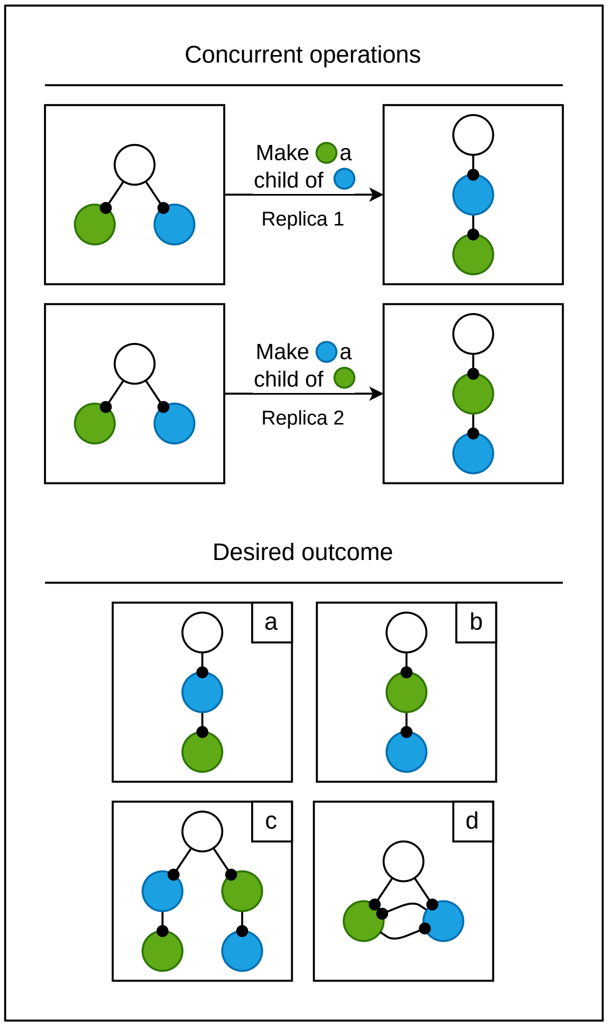

Figure 4. A case in which concurrent move operations take place for two nodes to become each others descendant, including the possible outcomes. The shown outcomes illustrate there is either an operation-discarding choice, a duplicating choice or a tree-structure breaking choice that must be made. Inspired by the illustrations designed by Kleppmann et al. [3]

The GumTree algorithm consists of a top-down phase and bottom-up phase to compute a set of mappings between both ASTs, while it leaves the algorithm to determine an edit script open to the implementor. The GumTree implementation for this work[6] to compute the edit script is based on the work of Chawathe et al. [1].

The class `MinimumEditScript` implements the *EditScript* algorithm [1], while the class `GumTreeAlgorithm` implements the *Gumtree* algorithm [2]. These implementations remain true to the specification of the author.

4 Implementing the Abstract Syntax Tree

The implementation of the conflict-free replicated abstract syntax tree can be viewed from two perspectives. On one hand, the implementation responsible for respecting the requirements of a CRDT, discussed in Section 4.1, and on the other hand the implementation for the underlying locally kept data structure, discussed in Section 4.2.

4.1 Implementing the CRDT

The mentioned motivation in Section 2 forms the basis of the `ConflicFreeReplicatedAst` class. Objects for this class are created with an object that implements the `Transmitter` trait. This trait guarantees the definition of a `publish` method to publish new edit scripts and a `subscribe` method to install a callback to receive incoming edit scripts.

The `ConflicFreeReplicatedAst` class implements the CRDT methods `update` for local replica changes, `query` to query for the local data structure and `merge` for incoming operations. Objects for this class locally keep a variable `operations` which contains the aggregation of all locally produced and received *edit scripts* that can be ordered by their timestamp.

Upon receiving local updates through `update`, it invokes the transmitter's `publish` method to propagate the changes to other replica's. The `query` method sorts the operations and folds them starting an empty AST, happening in a non-deterministic manner to guarantee the same set of operations across replica's results in the same AST. The `merge` method adds the incoming operations to the locally stored operations.

4.2 Implementing the local AST

The abstract syntax tree itself is implemented by the `HeadedAST` case class, which consists of an optional root (in case the AST is empty) and a mapping from identities to nodes, called its *header*. This class does not assume a type of identity used to identify nodes, as it is implemented through generics. This case class, as most of the codebase, is implemented in a functional style. The choice for doing so, as it was the first class on which work began, was to ease the feature to perform the rollback as through functional code the application of edits in the same order across all replica's would always yield the same outcome AST. Additionally, the code is easier to test.

[6]The authors of GumTree provided their implementation as a publicly available GitHub project, however, it is implemented using the Java language and is not developed with compilation for a web browser in mind.

The internal structure of the AST, consisting of a mapping and an optional root, would allow for easy serialization of the AST by serializing the root and the nodes. The nodes of the AST each implementing `SchemeNode` trait, covering generic syntax node operations such as querying for the parent, the depth of the tree and such. A more intuitive implementation for the AST would be to implement the tree structure as a root with pointers to other nodes (that in their turn point to other nodes) rather than keeping a mapping in the AST from identity to the nodes. Having opted use of this indirection through the AST *header* makes it trivial to serialize AST nodes, as they thus never consist of pointers to values in memory (which is critical in a distributed setting since replica's classically do not share a memory-space). Serialization is left to the BooPickle library such that it does not add to the complexity of the code, which is developed with high performance in mind [4].

The nodes of the Scheme-like language that has been implemented are limited to *numbers*, *strings*, *identifiers* and *expressions*. This set of nodes is relatively small but is sufficiently large to provide a prototype that covers all source-code edit operations for a language supporting primitive values and nodes with descendants, showing support for arbitrary nested setups.

Turning a textual representation of a program into its `HeadedAST` representation is made possible by the `Parser` object, which requires an identity generator function to assign each node with a new identity, which parses the programs using the grammar developed in combination with the FastParse library.

5 Integrated Demo

The project comes included with a working prototype. The prototype uses the `ConflictFreeReplicatedIntAst` class which extends the `ConflictFreeReplicatedAst` class. It provides concrete types for the `Identity` generic (as a pair of the replica identity combined with a unique integer) and the `EditIdentity` generic (as a pair of a Lamport clock in combination with a replica identity) as Listing 1 shows[7].

The `ConflictFreeReplicatedIntAst` class and the `ReplicatedIntOp` class are the dependencies used by the `CRDT_AST_ScalaJS` object, which is the interfacing file with the JavaScript codebase that makes the demo possible in a web browser.

The demo uses the Monaco Editor, which is the same editor that is used in the VS Code editor. Through a correct configuration during setup this enables syntax highlighting for Scheme-like languages. Throughout the lifetime of the code whenever the code is changed the AST is kept formatted in the browser to ensure all replica's with the same operations have the same view.

```
// file: ReplicatedIntOp.scala
case class NId(id: Int) extends AnyVal
case class LClock(count: Int) extends AnyVal
case class RId(identity: Int) extends AnyVal

// file: ConflictFreeReplicatedAst.scala
class ConflictFreeReplicatedAst[
    Identity,
    EditIdentity
](...){...}

// file: ConflictFreeReplicatedAst.scala
case class ConflictFreeReplicatedIntAst(...)
    extends ConflictFreeReplicatedAst[
        (RId, NId),
        (LClock, RId)
    ](...){...}
```

Listing 1. Code snippets across three files demonstrating how the `ConflictFreeReplicatedAst` is specialized for the demo. The identity of AST nodes is represented as a pair of a replica identity and a node identity. The identity of edit operations is represented as a pair of a Lamport clock and a replica identity.

To enable the propagation of operations for the operation-based CRDT, the demo uses the PeerJS library. This is a JavaScript library that allows setting up peer-to-peer connections between different clients on the web using the WebRTC standard. This network setup is used but can be interchanged with other configurations such as a client-server setup.

Through the use of the BooPickle library, the operations that need to be sent over the wire are serialized to bytes before they are transmitted to other peers. By doing so rather than sending textual representations the demo aims to maintain a low network load.

6 Tests

The project includes a test suite containing 53 tests, of which all succeed. For the given test suite, the IntelliJ Test Coverage Tool reports 100% code coverage[8] for the scala code, except the entry point for the JavaScript environment as this consist of 40 lines of JavaScript bindings. This broad testing suite paired with the high amount of code coverage aims to demonstrate the effort in verifying the correctness of the codebase.

An integration test that covers the case in which different operations reach replica's in different orders is present in the `"Test the replicated ASTs"` test that is present in the

[7]By having the case classes extend `AnyVal`, the compiler handles these as value classes. This allows for compile-time safety to not mix up types while keeping a low memory footprint by representing the values as integers at runtime.

[8]Note that to view this code coverage report, the IntelliJ feature to export code coverage to an HTML format was used, which shows that the sole methods that are not covered are the generated macro's originating from the pickling library to serialize the operations.

```
for (subset <- operations.subsets()) {
    val ordered = new ConflictFreeReplicatedAst()
    val shuffled = new ConflictFreeReplicatedAst()

    val operations = subset.toSeq

    ordered merge operations
    shuffled merge rnd.shuffle(operations)
    assert(ordered.query isomorphic shuffled.query)
}
```

Listing 2. A snippet of the tests (with technical details left out) that takes a set of operations, iterates through the powerset of these operations (thus taking into account operations that are temporarily out of order) and asserts that the application of both the ordered and the shuffled sequence of operations to an empty AST results in isomorphic ASTs.

`ConflictFreeReplicatedIntAstTest` class. A snippet of this test is reported in Listing 2. This listing shows how the iteration through the powerset of a set of operations applied to a newly constructed AST, both in ordered and randomly shuffled order, results in isomorphic outcome ASTs.

7 Future Work

Different improvements for the current implementation are discussed below.

- **Arbitrary order of operations.**
  A limitation for the current implementation is that the merge strategy performs an ordering of the concurrent operations in an arbitrary order (in the prototype this is based on the replica identity), though deterministic there is no meaning as to which replica performs the winning operations[9].
  An improvement hereto could be to assign developer roles through the user interface, allowing to sort the operations in an order that the hierarchically higher roles their operations are prioritized to win over other operations. An alternative improvement could be for the user interface to prompt the user to decide which action should dominate.
- **Merge granularity.**
  The merges take place on a granularity of happening on a per-node basis. Though sufficiently powerful, different merging strategies could be desired based on the types of nodes that are being affected, two examples are provided.
  For string literal nodes, two concurrent updates will only take into account the final operation after the sort to have taken place, while it would be more optimal to delegate the update operations to the string node to further apply the operations that are appropriate to merge text, similar to text editors.
  For literal set nodes, two concurrent additions of an equal value would duplicate the content, while this does not make sense for set nodes thus it would be the desired effect to prevent the double addition of the content.

[9]The mentioned 'no winning operation' statement holds only for concurrent operations on nodes that are already present on both replica's, say one replica updates a literal number to negative infinity while concurrently another replica updates the value to positive infinity, the result actual result will be an arbitrary operation based on the replica identities. For the addition of new nodes, no issues arise.

8 Conclusion

This work covers an alternative strategy to implement a real-time collaborative code editor to overcome the limitations that arise for traditional editors that do not reason about their content. It does so by modeling the underlying AST as a conflict-free replicated data type. The implementation covers a minimal set of nodes, demonstrating the capability of the merging strategy to resolve conflicts for different concurrent operations. The strategy is mainly based on the inclusion of a logical timestamp allowing for an ordering of events that allows the data structure to recompute the AST including newly received updates. Even so, certain optimizations are in order to improve the conflict resolution, either by inclusion of developer-roles or by shifting the merge granularity.

References

[1] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. 1996. Change Detection in Hierarchically Structured Information. *SIGMOD Rec.* 25, 2 (jun 1996), 493–504. https://doi.org/10.1145/235968.233366

[2] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[3] Martin Kleppmann, Dominic P. Mulligan, Victor B. F. Gomes, and Alastair R. Beresford. 2022. A Highly-Available Move Operation for Replicated Trees. *IEEE Transactions on Parallel and Distributed Systems* 33, 7 (2022), 1711–1724. https://doi.org/10.1109/TPDS.2021.3118603

[4] Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. 2013. Instant Pickles: Generating Object-Oriented Pickler Combinators for Fast and Extensible Serialization. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &amp; Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 183–202. https://doi.org/10.1145/2509136.2509547

[5] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.