Putting Order in Strong Eventual Consistency

Kevin De Porre[1], Florian Myter[1], Christophe De Troyer[1], Christophe Scholliers[2], Wolfgang De Meuter[1], and Elisa Gonzalez Boix[1]

[1] Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium
[2] Ghent University, Sint Pietersnieuwstraat 33, Gent, Belgium

Abstract. Conflict-free replicated data types (CRDTs) aid programmers develop highly available and scalable distributed systems. However, the literature describes only a limited portfolio of conflict-free data types and implementing custom ones requires additional knowledge of replication and consistency techniques. As a result, programmers resort to ad hoc solutions which are error-prone and result in brittle systems. In this paper, we introduce strong eventually consistent replicated objects (SECROs), a general-purpose data type for building available data structures that guarantee strong eventual consistency (SEC) without restrictions on the operations. To evaluate our solution we compare a real-time collaborative text editor built atop SECROs with a state-of-the-art implementation that uses JSON CRDTs. This comparison quantifies various performance aspects. The results show that SECROs are truly general-purpose and memory efficient.

Keywords: Distribution · Eventual consistency · Replicated data types

1 Introduction

According to the CAP theorem [4,5] distributed systems that are prone to partitions can only guarantee availability or consistency. This leads to a spectrum of distributed systems that ranges from highly available systems (AP) to strongly consistent systems (CP) with hybrid systems - that are partly AP and partly CP - in the middle. A substantial body of research has focused on techniques or protocols to propagate updates [7,16,19,20]. In this paper, we focus on *language abstractions* that ease the development of highly available and partition tolerant systems, the so-called AP systems.

A state-of-the-art approach towards high availability are conflict-free replicated data types (CRDTs) [19]. CRDTs rely on commutative operations to guarantee strong eventual consistency (SEC), a variation on eventual consistency that provides an additional *strong convergence* guarantee [3]. This avoids the need for synchronisation, yielding high availability and low latency.

The literature has proposed a portfolio of basic conflict-free data structures such as counters, sets, and linked lists [17,18,22]. However, advanced distributed

[3] Strong convergence states that correct replicas that received the same updates must be in an equivalent state.

systems require replicated data types that are tailored to the needs of the application. Consider, for example, a real-world collaborative text editor that represents documents as a balanced tree of characters, allowing for logarithmic time lookups, insertions, and deletions. To the best of our knowledge, the only tree CRDT has been proposed in [12]. In this approach, balancing the tree requires synchronising the replicas. However, this is not possible in AP systems as it implies giving up on availability.

When the current portfolio of CRDTs falls short, programmers can resort to two solutions. One is to manually engineer the data structure as a CRDT. This requires rethinking the data structure completely such that all operations commute. If the operations cannot be made commutative, programmers need to manually implement conflict resolution. This has shown to be error-prone and results in brittle systems [1,9,19]. Alternatively, programmers can use JSON CRDTs [9] or Lasp [14] to design custom CRDTs. JSON CRDTs let programmers arbitrarily nest linked lists and maps into new CRDTs, whereas Lasp supports functional transformations over existing CRDTs. However, these constructs are not general enough. Consider again the case of a collaborative text editor. Using lists and maps one cannot implement a balanced tree CRDT, nor can one derive a balanced tree from existing CRDTs.

In this paper, we explore a new direction which consists in devising a general-purpose language abstraction for high availability. We design a novel replicated data type called *strong eventually consistent replicated object* (SECRO). SECROs guarantee SEC by reordering *conflicting* operations in a way that solves the conflict. To find a conflict-free ordering of the operations, SECROs rely on application-specific information provided by the programmer through *concurrent pre and postconditions* defined over the operations of the SECRO. Our approach is based on the idea that conflict detection and resolution naturally depends on the semantics of the application [21].

We evaluate our approach by implementing a real-time collaborative text editor using SECROs and comparing it to a JSON CRDT implementation of the text editor, as proposed in [9]. We present various experiments that quantify the memory usage, execution time, and throughput of both implementations.

2 Strong Eventually Consistent Replicated Objects

In this section, we describe strong eventually consistent replicated objects from a programmer's perspective. All code snippets are in CScript [4], a JavaScript extension embodying our implementation of SECROs. We introduce the necessary syntax and features of CScript along with our explanation on SECROs.

2.1 SECRO data type

A SECRO is an object that implements an abstract data type and can be replicated to a group of devices. Like regular objects, SECROs contain state in the

[4] CScript is available at https://gitlab.com/iot-thesis/framework/tree/master

form of fields, and behaviour in the form of methods. It is not possible to directly access a SECRO's internal state. Instead, the methods defined by the SECRO need to be used. These methods form the SECRO's public interface. Methods can be further categorised in *accessors* (i.e. methods querying internal state) and *mutators* (i.e. methods updating the internal state).

As an example, consider the case of a collaborative text editor which organises documents as a balanced tree of characters [15, 22]. Listing 1.1 shows the structure of the `Document` SECRO. In order to create a new SECRO, programmers extend the `SECRO` abstract data type. Instead of implementing our own balanced tree data structure, we re-use an existing AVL tree data structure provided by the Closure library [5].

```
class Document extends SECRO {
  constructor(tree = new AvlTree((c1, c2) => c1.id - c2.id)) {
    this._tree = tree;
  }
  @accessor
  containsId(id) {
    const dummyChar = {char: '', id: id};
    return this._tree.contains(dummyChar);
  }
  @accessor
  generateId(prev) { /* see appendix */ }
  @accessor
  indexOf(char) {
    return this._tree.indexOf(char);
  }
  // serialisation methods
  tojson() {
    return this._tree; // AVL tree is serialisable
  }
  static fromjson(tree) {
    return new Document(tree);
  }
  // operations to manipulate the tree
  insertAfter(id, char) { /* see listing 1.2 */ }
  delete(id) { /* see listing 1.3 */ }
  // SECRO's state validators
  pre insertAfter(doc, args) {/*listing 1.2*/}
  post insertAfter(originalDoc, doc, args, newChar) {/*listing 1.2*/}
  post delete(originalDoc, doc, args, res) {/*listing 1.3*/}
}
Factory.registerAvailableType(Document);
```

Listing 1.1: Structure of the text editor.

The `Document` SECRO defines three accessors (`containsId`, `generateId` and `indexOf`) and two mutators (`insertAfter` and `delete`). `containsId` returns a boolean that indicates the presence or absence of a certain identifier in the

[5] https://developers.google.com/closure/library/

document tree. `generateId` uses a boundary allocation strategy [15] to compute stable identifiers based on the reference identifiers. Finally, `indexOf` returns the index of a character in the document tree. Note that side-effect free methods are annotated with `@accessor`, otherwise, CScript treats them as mutators.

The `Document` SECRO also defines methods to serialise and deserialise the document as it will be replicated over the network. Note that deserialisation creates a new replica of the `Document` SECRO. In order for the receiver to know the `Document` class, programmers must register their SECRO at the CScript *factory* (line 31).

Finally, the `Document` SECRO forwards `insertAfter` and `delete` operations on the text to the underlying AVL tree (as we describe later in Section 2.2). Besides the methods defined in the SECRO's public interface, programmers can also enforce application-specific invariants by associating concurrent preconditions and postconditions to the mutators (Lines 27 to 29). We say that pre and postconditions are *state validators*. State validators are used by the SECRO to order concurrent operations such that they do not violate any invariant. Next section further describes them.

2.2 State Validators

State validators let programmers define the data type's behaviour in the face of concurrency. State validators are declarative rules that are associated to mutators. Those rules express invariants over the state of the object which need to uphold in the presence of concurrent operations [6]. Behind the scenes, the replication protocol may interleave concurrent operations. From the programmer's perspective the only guarantee is that these invariants are upheld. State validators come in two forms:

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

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

In CScript, state validators are methods which are prefixed with the `pre` or `post` keyword, defining a pre or postcondition, respectively. To illustrate state validators we again consider the example of a collaborative text editor and present the implementation of the `insertAfter` and `delete` methods and their associated preconditions and postconditions. Listing 1.2 contains the `insertAfter`

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

operation. The `id` argument on Line 1 is the identifier of the reference character. On Line 2 the method generates a new stable identifier for the character it is inserting. Using this identifier the method creates a new character on Line 3. Finally, Lines 4 and 5 insert the character in the tree and return the newly added character. Lines 7 to 10 define a precondition on insert. The precondition is a method which has the same name as its associated operation and takes as parameters the object's current state followed by an array containing the arguments that are passed to its associated operation. In this case, `id` and `char` as passed to `insertAfter`. The precondition checks that the reference character exists (Line 9).

```
1  insertAfter(id, char) {
2      const newId   = this.generateId(id),
3            newChar = new Character(char, newId);
4      this._tree.add(newChar);
5      return newChar;
6  }
7  pre insertAfter(doc, args) {
8      const [id, char] = args;
9      return id === null || doc.containsId(id);
10 }
11 post insertAfter(originalDoc, newDoc, args, newChar) {
12     const [id, char]  = args,
13           originalChar = {char: "dummy", id: id};
14     return (id === null && doc._tree.contains(newChar)) ||
15             doc.indexOf(originalChar) < doc.indexOf(newChar);
16 }
```

Listing 1.2: Inserting a character in a tree-based text document.

Lines 11 to 16 define a postcondition for the `insertAfter` operation. Similar to preconditions, postconditions are defined as a method which has the same name as its associated operation (`insertAfter` in this case). However, they take 4 arguments: 1) the SECRO's initial state, 2) the state that results from applying the operation (`insertAfter`), 3) an array with the operation's arguments, and 4) the operation's return value (`newChar` in this case). This postcondition checks that the newly added character occurs at the correct position in the resulting tree, i.e. after the reference character that is identified by `id`. According to this postcondition any interleaving of concurrent character insertions is valid, e.g. two users may concurrently write "foo" and "bar" resulting in one of: "foobar","fboaor", etc. If the programmer only wants to allow the interleavings "foobar" and "barfoo" the SECRO must operate on the granularity of words instead of single character manipulations.

Listing 1.3 contains the implementation of the `delete` method and its associated postcondition. Lines 1 to 3 show that characters are deleted by removing them from the underlying AVL tree. Recall that the character's stable identifier uniquely identifies the character in the tree. Afterwards, the postcondition on Lines 4 to 7 ensures that the character no longer occurs in the tree.

```
1  delete(id) {
    return this._tree.remove(id);
}
post delete(originalDoc, doc, args, res) {
    const [id] = args;
    return !doc.containsId(id);
}
```

Listing 1.3: Deleting a character from a tree-based text document.

Notice that preconditions are less expressive than postconditions but, they avoid unnecessary computations by rejecting invalid states prior to the execution of the operation. Preconditions are also useful to prevent operations from running on a corrupted state, thus improving the system's robustness.

3 SECRO's Replication Protocol

A SECRO is a general-purpose language abstraction that guarantees SEC, i.e. eventual consistency and strong convergence. To provide this guarantee SECROs implement a dedicated optimistic replication protocol. For the purpose of this paper, we describe the protocol in pseudocode [7].

SECRO's protocol propagates update operations to all replicas. In contrast to CRDTs, the operations of a SECRO do not necessarily commute. Therefore, the replication protocol totally orders the operations at all replicas. This order may not violate any of the operations' pre or postconditions.

For the sake of simplicity we assume a causal order broadcasting mechanism without loss of generality, i.e. a communication medium in which messages arrive in an order that is consistent with the happened-before relation [10]. Note that even though we rely on causal order broadcasting, concurrent operations arrive in arbitrary orders at the replicas.

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

The existence of a valid execution cannot be guaranteed if pre and postconditions contradict each other. It is the programmer's responsibility to provide correct pre and postconditions.

The replication protocol provides the following guarantees:

[7] The implementation is part of CScript and can be found at https://gitlab.com/iot-thesis/framework/tree/master/src/Application/CRDTs/SECRO

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

The operation histories of replicas may grow unboundedly as they perform operations. In order to alleviate this issue we allow for replicas to periodically *commit* their state. Concretely, replicas maintain a *version number*. Whenever a replica commits, it clears its operation history and increments its version number. The replication protocol then notifies all other replicas of this commit, which adopt the committed state and also empty their operation history. All subsequent operations received by a replica which apply to a previous version number are ignored. As we explain in Section 3.1, the commit operation does not require synchronising the replicas and thus does not affect the system's availability. However, commits come at the price of certain operations being dropped for the sake of bounded operation history size.

3.1 Algorithm

We now detail our replication protocol which makes the following assumptions:

- Each node in the network may contain any number of replicas of a SECRO.
- Nodes maintain vector clocks to timestamp the operations of a replica.
- Nodes are able to generate globally unique identifiers using lamport clocks.
- Reading the state of a replica happens side-effect free and mutators solely affect the replica's state (i.e. the side effects are confined to the replica itself).
- Eventually, all messages arrive, i.e. reliable communication: no message loss nor duplication (e.g., TCP/IP).
- There are no byzantine failures, i.e. no malicious nodes.

A replica r is a tuple $r = (v_i, s_0, s_i, h, id_c)$ consisting of the replica's version number v_i, its initial state s_0, its current state s_i, its operation history h, and the id of the latest commit operation id_c. A mutator m is represented as a tuple $m = (o, p, a)$ consisting of the update operation o, precondition p, and postcondition a. We denote that a mutation m_1 happened before m_2 using $m_1 \prec m_2$. Similarly, we denote that two mutations happened concurrently using $m_1 \parallel m_2$. Both relations are based on the clocks carried by the mutators [8].

We now discuss in detail the three kinds of operations that are possible on replicas: reading, mutating, and committing state.

Reading Replicas Reading the value of a replica (v_i, s_0, s_i, h, id_c) simply returns its latest local state s_i.

Mutating Replicas When a mutator $m = (o, p, a)$ is applied to a replica a *mutate* message is broadcast to all replicas. Such a message is an extension of the mutator (o, p, a, c, id) which additionally contains the node's logical clock time c and a unique identifier id.

ALGORITHM 1: Handling *mutate* messages

```
    arguments: A mutate message m = $(o, p, a, c, id)$, a replica = $(v_i, s_0, s_i, h, id_c)$
 1  $h' = h \cup \{m\}$
 2  for $ops \in LE(sort_{>>}(h'))$ do
 3      $s_i' = s_i$
 4      $pre = 0$
 5      $post = 0$
 6      for $m \in ops$ do
 7          $concurrentClosure = TC(m, h') \cup \{m\}$
 8          for $m' = (o, p, a, c, id) \in concurrentClosure$ do
 9              if $p(s_i')$ then
10                  $pre \mathrel{+}= 1$
11                  $s_i' = o(s_i')$
12              end
13          end
14          for $m' = (o, p, a, c, id) \in concurrentClosure$ do
15              if $a(s_i')$ then
16                  $post \mathrel{+}= 1$
17              end
18          end
19          $ops = ops \setminus concurrentClosure$
20      end
21      if $pre == |ops| \wedge post == |ops|$ then
22          return $(v_i, s_0, s_i', h', id_c)$
23      end
24  end
    // throw faulty program exception
```

As mentioned before, operations on SECROs do not need to commute by design. Since operations are timestamped with logical clocks they exhibit a partial order. Algorithm 1 governs the replicas' behaviour to guarantee SEC by ensuring that all replicas execute the same valid ordering of their operation history.

Algorithm 1 starts when a replica receives a *mutate* message. The algorithm consists of two parts. First, it adds the *mutate* message to the operation history, sorts the history according to the >> total order, and generates all *linear extensions* of the replica's sorted history (see Lines 1 and 2). We say that $m_1 = (o_1, p_1, a_1, c_1, id_1) >> m_2 = (o_2, p_2, a_2, c_2, id_2)$ iff $c_1 \succ c_2 \vee (c_1 \parallel c_2 \wedge id_1 > id_2)$. The generated linear extensions are all the permutations of h' that respect the partial order defined by the operations' causal relations. Since replicas deterministically compute linear extensions and start from the same sorted operation history, all replicas generate the same sequence of permutations.

Second, the algorithm searches for the first *valid* permutation. In other words, for each operation within such a permutation the algorithm checks that the preconditions (Lines 8 to 13) and postconditions (Lines 14 to 18) hold. Remember that postconditions are checked only after all concurrent operations executed since they happened independently (e.g. during a network partition) and may thus conflict. For this reason, Line 7 computes the transitive closure of concurrent operations [8] for every operation in the linear extension.

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

Since the "is concurrent" relation is not transitive, one might wonder why we consider operations that are not directly concurrent. To illustrate this, consider a replica r_1 that executes operation o_1 followed by o_2 ($o_1 \prec o_2$) while concurrently replica r_2 executes operation o_3 ($o_3 \parallel o_1 \land o_3 \parallel o_2$). Since o_3 may affect both o_1 and o_2 we take into account all three operations. This corresponds to the transitive closure $\{o_1, o_2, o_3\}$. We refer the reader to Appendix A for a proof that no operation can break this transitive closure of concurrent operations.

Finally, the algorithm returns the replica's updated state as soon as a valid execution is found, otherwise, it throws an exception.

Committing Replicas In a nutshell, commit clears a replica's operation history h, increments the replica's version and updates the initial state s_0 with the replica's current state s_i. This avoids unbounded growth of operation histories, but operations concurrent with the commit will be discarded [9].

When a replica is committed a *commit* message is broadcast to all replicas (including the committed one). This message is a quadruple $(s_i, v_i, clock, id)$ containing the committed state, the replica's version number, the current logical clock time, and a unique id.

ALGORITHM 2: Handling *commit* messages

```
arguments: A commit message = $(s_c, v_c, clock, id)$, a replica = $(v_i, s_0, s_i, h, id_c)$
1 if $v_c = v_i$ then
2     return $(v_i + 1, s_c, s_c, \emptyset, id)$
3 end
4 if $v_c = v_i - 1 \land id < id_c$ then
5     return $(v_i, s_c, s_c, \emptyset, id)$
6 end
```

To ensure that replicas converge in the face of concurrent commits we design commit operations to commute. As a result, commit does not compromise availability. Algorithm 2 dictates how replicas handle *commit* messages. The algorithm distinguishes between two cases. First, the commit operation commits the current state (see Line 1). The replica's version is incremented, its initial and current state are set to the committed state, the operation history is cleared and the id of the last performed commit is updated. Second, the commit operation commits the previous state (see Line 4). This means that the commit operation applies to the previous version v_{i-1}. As a result, the newly received commit operation is concurrent with the last performed commit operation (i.e. the one that caused the replica to update its version from v_{i-1} to v_i). To ensure convergence, replicas perform the commit operation with the smallest ID. This ensures that the order in which commits are received is immaterial and hence that commit operations commute. Note that the algorithm does not need to tackle the case of committing an elder state since it cannot happen under the assumption of causal order broadcasting.

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

4 Evaluation

We now compare our novel replicated data type to JSON CRDTs, a state-of-the-art approach providing custom CRDTs built atop lists and maps. We perform a number of experiments which quantify the memory usage, execution time and throughput of the collaborative text editor. We implemented it twice in JavaScript, once using SECROs [10] and once using JSON CRDTs [11]. The JSON CRDT implementation uses a list to represent text documents. The SECRO implementation comes in two variants: one that uses a list and one that uses a balanced tree (described in Section 2).

Note that SECROs are designed to ease the development of custom replicated data types guaranteeing SEC. Hence, our goal is not to outperform JSON CRDTs, but rather to evaluate the practical feasibility of SECROs.

4.1 Methodology

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

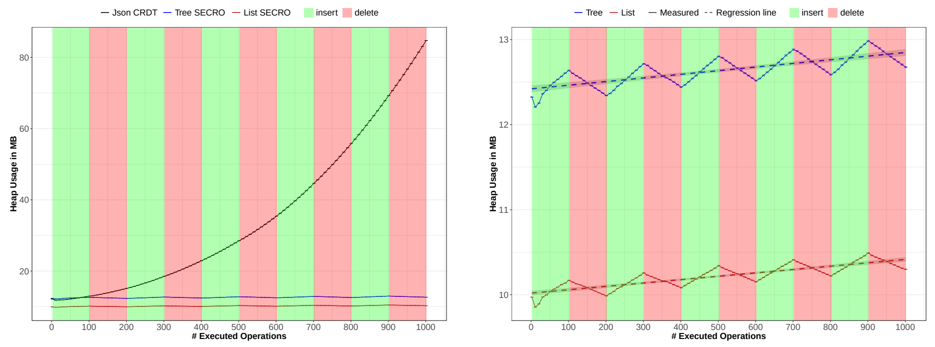
To get statistically sound results we repeat each benchmark at least 30 times, yielding a minimum of 30 samples per measurement. Each benchmark starts with a number of warmup rounds to minimise the effects of program initialisation. Furthermore, we disable NodeJS' just-in-time compiler optimisations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

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

4.2 Memory Usage

To compare the memory usage of the SECRO and JSON CRDT text editors, we perform an experiment in which 1000 operations are executed on each text editor. We continuously alternate between 100 character insertions followed by deletions of those 100 characters. We force garbage collection after each operation [12], and measure the heap usage. The resulting measurements are shown in Figure 1. Green and red columns indicate character insertions and deletions respectively.

[10] CScript code presented in Section 2 compiles to JavaScript and runs atop NodeJS.
[11] The implementations are available at https://gitlab.com/iot-thesis/framework/tree/master
[12] Forcing garbage collection is needed to get the real-time memory usage. Otherwise, the memory usage keeps growing until garbage collection is triggered.



(a) Comparison between the memory usage of the SECRO and JSON CRDT text editors.

(b) Comparison between the list and tree implementations of the SECRO text editor.

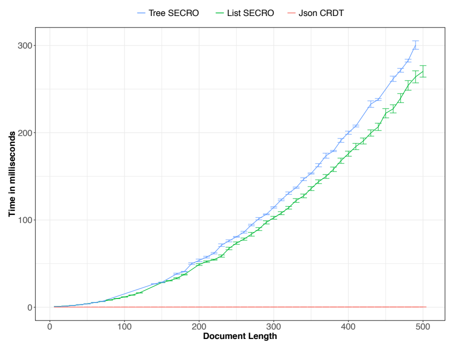
Fig. 1: Memory usage of the collaborative text editors. Error bars represent the 95% confidence interval for the average taken from 30 samples. These experiments are performed on a single worker node of the cluster.

Figure 1a confirms our expectation that the SECRO implementations are more memory efficient than the JSON CRDT one. The memory usage of the JSON CRDT text editor grows unbounded since CRDTs cannot delete characters, but merely mark them as deleted using tombstones. Conversely, SECROs support true deletions by reorganising concurrent operations in a non-conflicting order. Hence, all 100 inserted characters are deleted by the following 100 deletions. This results in lower memory usage.

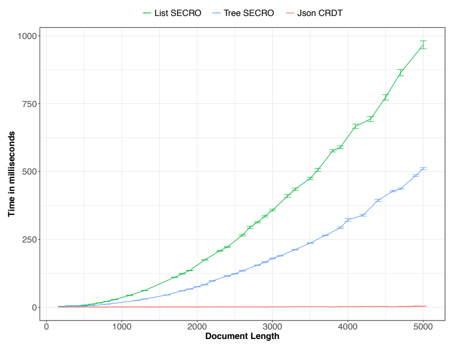
Figure 1b compares the memory usage of the list and tree-based implementations using SECROs. We conclude that the tree-based implementation consumes more memory than the list implementation. The reason is that nodes of a tree maintain pointers to their children, whereas nodes of a singly linked list only maintain a single pointer to the next node. Interestingly, we observe a staircase pattern. This pattern indicates that memory usage grows when characters are inserted (green columns) and shrinks when characters are deleted (red columns). Overall, memory usage increases linearly with the number of executed operations, even though we delete the inserted characters and commit the replica after each operation. Hence, SECROs cause a small memory overhead for each executed operation. This linear increase is shown by the dashed regression lines.

4.3 Execution Time

We now benchmark the time it takes to append characters to a text document. Although this is not a realistic edition pattern, it showcases the worst case performance. From Figure 2a we notice that the SECRO versions exhibit a quadratic performance, whereas the JSON CRDT version exhibits a linear performance. The reason for this is that reordering the SECRO's history (see Algorithm 1 in Section 3.1) induces a linear overhead on top of the operations themselves. Since insert is also a linear operation, the overall performance of the text editor's insert operation is quadratic. To address this performance overhead the replica needs to be committed. The effect of commit on the execution time of insert operations is depicted in Appendix B.



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



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

Fig. 2: Execution time of character insertions in the collaborative text editors. Replicas are never committed. Error bars represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection are discarded.

Figure 2a also shows that the SECRO implementation that uses a linked list is faster than its tree-based counterpart. To determine the cause of this counterintuitive observation, we measure the different parts that make up the total execution time:

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

Figures 5a and 5b in Appendix C depict the detailed execution time for the list and tree implementations respectively. The results show that the total execution time is dominated by the copy time. We observe that the tree implementation spends more time on copying the document than the list implementation. The reason being that copying a tree entails a higher overhead than copying a linked list as more pointers need to be copied. Furthermore, the tree implementation spends considerably less time executing operations, preconditions and postconditions, than the list implementation. This results from the fact that the balanced tree provides logarithmic time operations.

Unfortunately, the time overhead incurred by copying the document kills the speedup we gain from organising the document as a tree. This is because each insertion inserts only a single character but requires the entire document to be copied. To validate this hypothesis, we re-execute the benchmark shown in Figure 2a but insert 100 characters per operation. Figure 2b shows the resulting execution times. As expected, the tree implementation now outperforms the list implementation. This means that the speedup obtained from 100 logarithmic insertions exceeds the copying overhead induced by the tree. In practice, this means that single character manipulations are too fine-grained. Manipulating entire words, sentences or even paragraphs is more beneficial for performance.

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

4.4 Throughput

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

Figure 3 depicts how the throughput of the list-based text editor varies in function of the load. We observe that the SECRO text editor scales up to 50 concurrent operations, at which point it reaches its maximal throughput. Afterwards, the throughput quickly degrades. On the other hand, the JSON CRDT implementation achieves a higher throughput than the SECRO version under high loads (100 concurrent operations and more). Hence, the JSON CRDT text editor scales better than the SECRO text editor, but SECROs are general-purpose which allowed us to organise documents as balanced trees of characters.

5 Related Work

We now discuss work that is closely related to the ideas presented in this paper. Central to SECROs is the idea of employing application-specific information to reorder conflicting operations. Bayou [21] was the first system to use application-level semantics for conflict resolution by means of merge procedures provided by users. Our work, however, does not require manual resolution of conflicts.

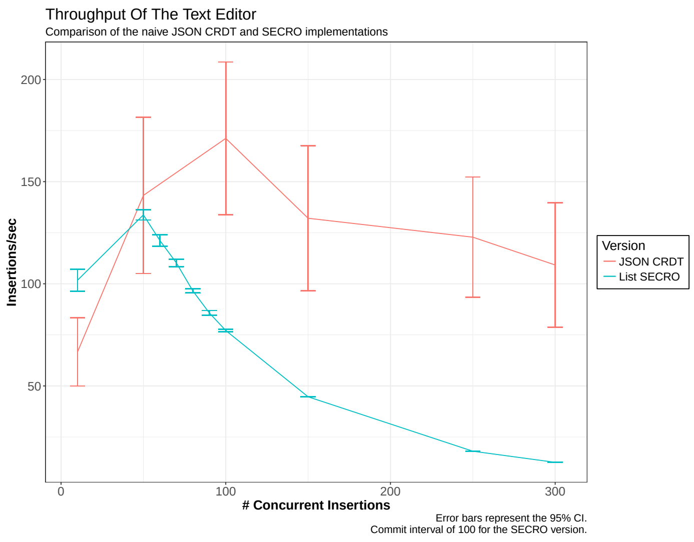


Fig. 3: Throughput of the list-based SECRO and JSON CRDT text editors, in function of the number of concurrent operations. The SECRO version committed the document replica at a commit interval of 100. Error bars represent the 95% confidence interval for the average of 30 samples.

Instead, programmers only need to specify the invariants the application should uphold in the face of concurrent updates, and the underlying update algorithm deterministically orders operations.

Within the CRDT literature, the research on JSON CRDTs [9] is the most closely related to our work. JSON CRDTs aim to ease the construction of CRDTs by hiding the commutativity restriction that traditionally applies to the operations. Programmers can build new CRDTs by nesting lists and maps in arbitrary ways. The major shortcoming is that nesting lists and maps does not suffice to implement arbitrary replicated data types. Hence, JSON CRDTs are not truly general-purpose as opposed to SECROs.

Lasp [14] is the first distributed programming language where CRDTs are first-class citizens. New CRDTs are defined through functional transformations over existing ones. In contrast, SECROs are not limited to a portfolio of existing data types that can be extended. Any existing data structure can be turned into a SECRO by associating state validators to the operations.

Besides CRDTs, cloud types [6] are high-level data types that can be replicated over the network. Similar to SECROs, cloud types do not impose restrictions on the operations of the replicated data type. However, cloud types hardcode how to merge updates coming from different replicas of the same type. As such, programmers have no means to customise the merge process of cloud types to fit the application's semantics. Instead, they are bound to implement a new cloud type and the accompanying merge procedure that fits the application. Hence, conflict resolution needs to be manually dealt with.

Some work has considered a hybrid approach offering SEC for commutative operations, and requiring strong consistency for non-commutative ones [2, 3]. There are some similarities to SECROs as they employ application-specific invariants to classify operations as safe or unsafe under concurrent execution. In this work, unsafe operations are synchronised while SECROs reorder unsafe operations as to avoid conflicts without giving up on availability. Partial Order-Restrictions (PoR) consistency [13] uses application-specific restrictions over operations but cannot guarantee convergence nor invariant preservation since these properties depend on the restrictions over the operations specified by the programmer.

6 Conclusion

In this work, we propose strong eventually consistent replicated objects (SECROs), a data type that guarantees SEC without imposing restrictions on the operations. SECROs do not avoid conflicts by design, but instead compute a global total order of the operations that is conflict-free, without synchronising the replicas. To this end, SECROs use *state validators*: application-specific invariants that determine the object's behaviour in the face of concurrency.

To the best of our knowledge, SECROs are the first approach to support truly general-purpose replicated data types while still guaranteeing SEC. By specifying state validators arbitrary data types can be turned into highly available replicated data types. This means that replicated data types can be implemented similarly to their sequential local counterpart, with the addition of preconditions and postconditions to define concurrent semantics. We showcase the flexibility of SECROs through the implementation of a collaborative text editor that stores documents as a tree of characters. The implementation re-uses a third-party AVL tree and turns into a replicated data type using SECROs.

We compared our SECRO-based collaborative text editor to a state-of-the-art implementation that uses JSON CRDTs. The benchmarks reveal that SECROs efficiently manage memory, whereas the memory usage of JSON CRDTs grows unbounded. Time complexity benchmarks reveal that SECROs induce a linear time overhead which is proportional to the size of the operation history. Performance wise, SECROs can be competitive to state-of-the-art solutions if committed regularly.

Acknowledgments

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

References

1. Almeida, P.S., Shoker, A., Baquero, C.: Efficient State-based CRDTs by Delta-Mutation. In: Bouajjani, A., Fauconnier, H. (eds.) Int. Conference on Networked Systems. pp. 62–76. Springer-Verslag, Agadir, Morocco (2015)
2. Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., Preguiça, N., Najafzadeh, M., Shapiro, M.: Putting Consistency Back into Eventual Consistency. In: 10th European Conference on Computer Systems. pp. 6:1–6:16. EuroSys '15 (2015)
3. Balegas, V., Li, C., Najafzadeh, M., Porto, D., Clement, A., Duarte, S., Ferreira, C., Gehrke, J., Leitao, J., Preguiça, N., et al.: Geo-Replication: Fast If Possible, Consistent If Necessary. IEEE Data Engineering Bulletin **39**(1), 12 (2016)
4. Brewer, E.: Towards Robust Distributed Systems. In: 19th Annual ACM Symp. on Principles of Distributed Computing. p. 7. PODC '00 (2000)
5. Brewer, E.: CAP Twelve Years Later: How the "Rules" Have Changed. Computer **45**(2), 23–29 (2012)
6. Burckhardt, S., Fähndrich, M., Leijen, D., Wood, B.P.: Cloud Types for Eventual Consistency. In: 26th European Conference on Object-Oriented Programming. pp. 283–307. ECOOP'12, Springer-Verlag, Berlin, Heidelberg (2012)
7. Burckhardt, S., Leijen, D., Protzenko, J., Fähndrich, M.: Global Sequence Protocol: A Robust Abstraction for Replicated Shared State. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming (ECOOP'15). vol. 37, pp. 568–590. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015)
8. de Juan-Marín, R., Decker, H., Armendáriz-Íñigo, J.E., Bernabéu-Aubán, J.M., Muñoz-Escoí, F.D.: Scalability approaches for causal multicast: a survey. Computing **98**(9), 923–947 (2016)
9. Kleppmann, M., Beresford, A.R.: A Conflict-Free Replicated JSON Datatype. IEEE Trans. on Parallel and Distributed Systems **28**(10), 2733–2746 (2017)
10. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM **21**(7), 558–565 (1978)
11. Lamport, L.: The Temporal Logic of Actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (May 1994)
12. Letia, M., Preguiça, N., Shapiro, M.: CRDTs: Consistency without concurrency control. Tech. rep., INRIA, Rocquencourt, France (2009), rR-6956
13. Li, C., Preguiça, N., Rodrigues, R.: Fine-grained consistency for geo-replicated systems. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). pp. 359–372. USENIX Association, Boston, MA (2018)
14. Meiklejohn, C., Van Roy, P.: Lasp: A Language for Distributed, Coordination-free Programming. In: 17th Int. Symp. on Principles and Practice of Declarative Programming. pp. 184–195. PPDP '15 (2015)
15. Nédelec, B., Molli, P., Mostefaoui, A., Desmontils, E.: LSEQ: An Adaptive Structure for Sequences in Distributed Collaborative Editing. In: Proc. of the 2013 ACM Symposium on Document Engineering. pp. 37–46. DocEng '13, Florence, Italy (2013)
16. Petersen, K., Spreitzer, M.J., Terry, D.B., Theimer, M.M., Demers, A.J.: Flexible Update Propagation for Weakly Consistent Replication. In: 16th ACM Symp. on Operating Systems Principles. pp. 288–301. SOSP '97 (1997)
17. Roh, H.G., Jeon, M., Kim, J.S., Lee, J.: Replicated Abstract Data Types: Building Blocks for Collaborative Applications. Journal of Parallel and Distributed Computing **71**(3), 354–368 (2011)

18. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA (Jan 2011)
19. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free Replicated Data Types. In: Défago, X., Petit, F., Villain, V. (eds.) 13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems. pp. 386–400. SSS'11, Springer-Verslag, Grenoble, France (2011)
20. Sun, C., Ellis, C.: Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In: Proc. of the 1998 ACM Conference on Computer Supported Cooperative Work. pp. 59–68. CSCW '98 (1998)
21. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In: Jones, M.B. (ed.) 15th ACM Symp. on Operating Systems Principles. pp. 172–182. SOSP '95 (1995)
22. Weiss, S., Urso, P., Molli, P.: Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. IEEE Trans. on Parallel and Distributed Systems **21**(8), 1162–1174 (Aug 2010)

A Proof: Operations Cannot Break the Transitive Closure of Concurrent Operations

Recall from Algorithm 1 in Section 3.1 that checking preconditions and postconditions requires computing the transitive closure of concurrent operations. We now formally define the transitive closure of concurrent operations and prove that operations cannot break this closure.

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

Definition 2. *Two operations m_1 and m_2 are concurrent iff neither one happened before the other [11]: $m_1 \parallel m_2 \iff m_1 \not\prec m_2 \wedge m_2 \not\prec m_1$.*

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

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

Definition 5. *An operation m happened before a set of operations T iff it happened before every operation of the set: $m \prec T \iff \forall m' \in T : m \prec m'$.*

Definition 6. *An operation m happened after a set of operations T iff it happened after all operations of the set: $T \prec m \iff \forall m' \in T : m' \prec m$.*

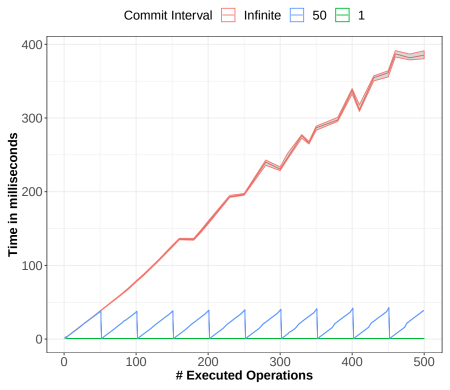
Definition 7. *A set of operations T_1 happened before a set of operations T_2 iff every operation from T_1 happened before every operation of T_2: $T_1 \prec T_2 \iff \forall m_1 \in T_1\, \forall m_2 \in T_2 : m_1 \prec m_2$*

Theorem 1. *For any operation m' and any non-empty transitive closure $TC(m,h)$ it holds that $m' \in TC(m,h) \vee m' \prec TC(m,h) \vee TC(m,h) \prec m'$.*

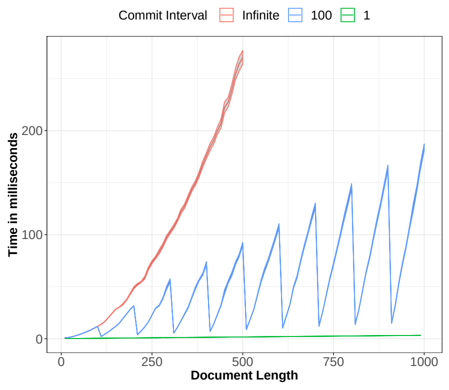
Proof. Proof by contradiction.
Assume that an operation m' exists for which Theorem 1 does not hold: $\exists m' : m' \notin TC(m,h) \wedge m' \not\prec TC(m,h) \wedge TC(m,h) \not\prec m'$. This means that operation m' breaks the concurrent transitive closure into two disjoint sets of operations: $\exists T_1 \exists T_2 : T_1 \prec m' \wedge m' \prec T_2$ where $T_1 = \{m_1, m_2, \ldots, m_i\} \subset TC(m,h)$ and $T_2 = \{m_{i+1}, \ldots, m_n\} \subset TC(m,h)$ and $T_1 \cap T_2 = \emptyset$ and $T_1 \cup T_2 = TC(m,h)$. Then by transitivity of the happened-before relation ($\prec$) we find that $T_1 \prec T_2$. This leads to a contradiction since we know that $T_1 \in TC(m,h) \wedge T_2 \in TC(m,h) \implies \exists m_i \in T_1 \exists m_j \in T_2 : m_i \parallel m_j$, i.e., there must be a link $m_i \parallel m_j$ between T_1 and T_2. Therefore, T_1 cannot have happened before T_2. □

B The Effect of Commit on the Execution Time

In this appendix, we present two benchmarks. The first quantifies the performance overhead of SECROs that results from reordering the operation history. The second illustrates the effect of commit on the execution time of the collaborative text editor and how commit improves its performance.



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



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

Fig. 4: Execution time of SECROs for different commit intervals, performed on a single worker node of the cluster. Error bands represent the 95% confidence interval for the average taken from a minimum of 30 samples. Samples affected by garbage collection were discarded.

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

Figure 4a depicts the execution time of the aforementioned constant time operation. If we do not commit the replica (red curve), the operation's execution time increases linearly with the number of operations. Hence, SECROs induce a linear overhead. This results from the fact that the replica's operation history grows with every operation. Each operation requires the replica to reorganise the history. To this end, the replica generates linear extensions of the history until a valid ordering of the operations is found (see Algorithm 1 in Section 3.1). Since we defined no preconditions or postconditions, every order is valid. The replica thus generates exactly one linear extension and validates it. To validate the ordering, the replica executes each operation. Therefore, the operation's execution time is linear to the size of the operation history.

As mentioned previously, commit implies a trade-off between concurrency and performance. Small commit intervals lead to better performance but less concurrency, whereas large commit intervals support more concurrent operations at the cost of performance. Figure 4a illustrates this trade-off. For a commit interval of 50 (blue curve), we observe a sawtooth pattern. The operation's execution time increases until the replica is committed, whereafter it falls back to its initial execution time. This is because *commit* clears the operation history. When choosing a commit interval of 1 (green curve), the replica is committed after every operation. Hence, the history contains a single operation and does not need to be reorganised. This results in a constant execution time.

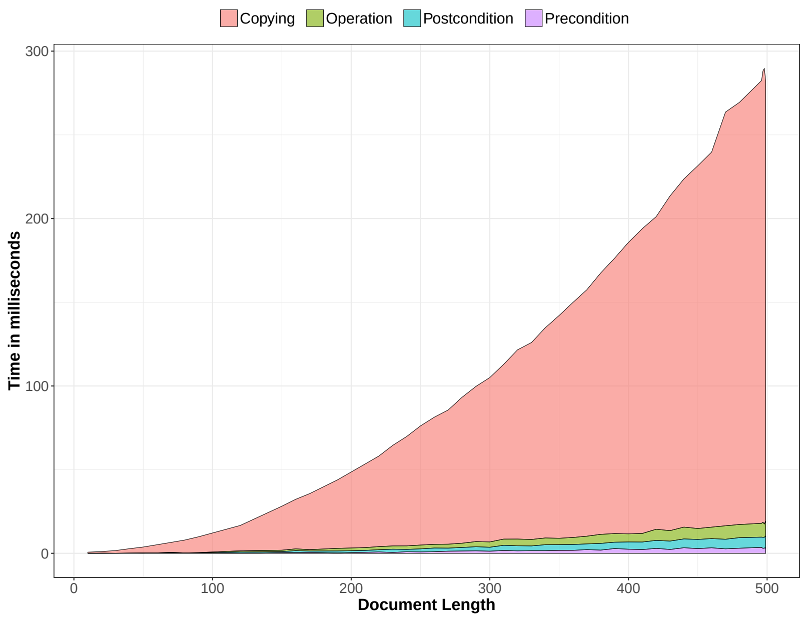
We now analyse the execution time of insert operations on the collaborative text editor. Figure 4b shows the time it takes to append a character to a text document in function of the document's length, for various commit intervals. If we do not commit the replica (red curve), append exhibits a quadratic execution time. This is because the SECRO induces a linear overhead and append is a linear operation. Hence, append's execution time becomes quadratic. For a commit interval of 100 (blue curve) we again observe a sawtooth pattern. In contrast to Figure 4a the peaks increase linearly with the size of the document, since append is a linear operation. If we choose a commit interval of 1 (green curve) we get a linear execution time. This results from the fact that we do not need to reorganise the replica's history. Hence, we execute a single append operation.

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

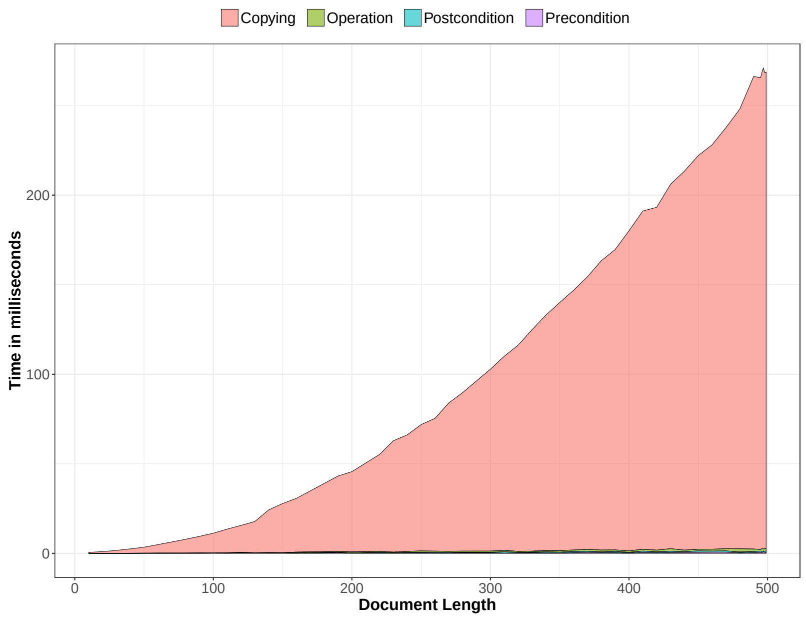
C Detailed Execution Time

In this appendix we show the detailed execution time of character insertions in the list and tree versions of the collaborative text editor. This is a breakdown of the green and blue curves respectively in Figure 2a. The replica is never committed. The plotted execution time is the average taken from a minimum of

30 samples. Samples affected by garbage collection are discarded. The complete explanation can be found in Section 4.3.



(a) List implementation



(b) Tree implementation