A coordination-free, convergent, and safe replicated tree

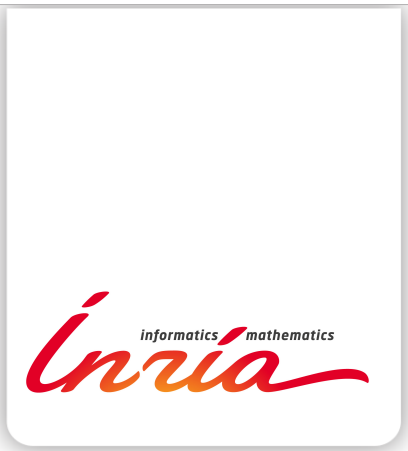
Sreeja Nair, Filipe Meirim, Mário Pereira, Carla Ferreira, Marc Shapiro

A coordination-free, convergent, and safe replicated tree

Sreeja S. Nair, Filipe Meirim, Mário Pereira, Carla Ferreira, Marc Shapiro

RESEARCH REPORT
N° 9395
February 2021
Project-Teams DELYS

ISSN 0249-6399 ISRN INRIA/RR--9395--FR+ENG



A coordination-free, convergent, and safe replicated tree

Sreeja S. Nair, Filipe Meirim, Mário Pereira, Carla Ferreira, Marc Shapiro

Project-Teams DELYS

Research Report n° 9395 — February 2021 — 24 pages

Abstract: The tree is an essential data structure in many applications. In a distributed application, such as a distributed file system, the tree is replicated. To improve performance and availability, different clients should be able to update their replicas concurrently and without coordination. Such concurrent updates converge if the effects commute, but nonetheless, concurrent moves can lead to incorrect states and even data loss. Such a severe issue cannot be ignored; ultimately, only one of the conflicting moves may be allowed to take effect. However, as it is rare, a solution should be lightweight. Previous approaches would require preventative cross-replica coordination, or totally order move operations after-the-fact, requiring roll-back and compensation operations.

In this paper, we present a novel replicated tree that supports coordination-free concurrent atomic moves, and provably maintains the tree invariant. Our analysis identifies cases where concurrent moves are inherently safe, and we devise a lightweight, coordination-free, rollback-free algorithm for the remaining cases, such that a maximal safe subset of moves takes effect.

We present a detailed analysis of the concurrency issues with trees, justifying our replicated tree data structure. We provide mechanized proof that the data structure is convergent and maintains the tree invariant. Finally, we compare the response time and availability of our design against the literature.

Key-words: Distributed data structures, Conflict-free Replicated Data Type, Formal verification

RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Un arbre répliqué, convergent et sûr sans coordination

Résumé : L'arbre est une structure de données essentielle. Quand l'application est distribuée, par exemple dans un système de fichiers distribué, l'arbre est répliqué. Pour améliorer les performances et la disponibilité, les différents clients doivent pouvoir mettre à jour leurs répliques simultanément et sans coordination. Celles-ci convergent si les mises à jour commutent entre elles ; néanmoins, même dans ce cas, des opérations "move" concurrentes peuvent conduire à des états incorrects, et même à la perte de données. Au bout du compte, entre deux oprations "move" en conflit, seul l'une des deux peut être autorisée à prendre effet. Cependant, comme ce cas est rare, la solution doit être légère. Les approches précédentes nécessitaient une coordination préventive des répliques, ou des retours en arrière à posteriori.

Dans cet article, nous prsentons un nouvel arbre répliqué, qui met en uvre une opération "move" atomique sans coordination, et dont nous prouvons qu'il maintient l'invariant d'arbre. Notre analyse identifie les cas oú les "move" concurrents sont intrinsèquement sûrs, et proposons un algorithme léger, sans coordination et sans retour-arrière, pour les autres cas, de sorte qu'un sous-ensemble maximal et sûr de "move" prenne effet.

Nous présentons une analyse détaillée des problèmes de cohérence dans les arbres. Nous fournissons une preuve mécanisée que la structure des données est convergente et maintient l'invariant d'arbre. Enfin, nous comparons le temps de rponse et la disponibilit de notre concept à la littérature.

Mots-clés : Structures de données distribues, CRDT, Vérification formelle

1 Introduction

Concurrent data structures are an important programming abstraction; designing concurrent data structures with non-trivial properties is complex. The tree data structure is used in many applications. For instance, a file system is a tree of directories and files. A move (or rename) operation transfers a subtree atomically by changing its parent. Similarly, a rich text editor maintains a DOM tree of blocks with attributes. Text editing modifies the tree structure; in particular a *drag and drop* can move a subtree from one parent to another.

A tree has a particularly strong structural invariant: nodes are unique, there is a single root, each node has a single parent and has a path to the root, and the child-parent graph is acyclic.

Much current work in concurrent data structure design focuses on lock-free or wait-free coordination using primitives such as compare-and-swap (CAS). However, in a distributed and replicated setting, even CAS is too strong. Consider for instance a file system replicated to several locations over the globe, or through a mobile network. Network latency between continents can be anywhere between 0.1 and 0.5 seconds; the mobile network may disconnect completely. To ensure availability, a user of the file system must be able to update a replica locally, and update *without coordinating at all* with other replicas. Replicas converge eventually by exchanging their updates asynchronously.

It is a major challenge to maintain safety in this context; specifically, in this case, to maintain the tree structure. Concurrent atomic moves (also called renames in a file system) are especially problematic [1]. Consider for instance a tree composed of the root and children a and b. One replica moves a underneath b, while concurrently (without coordination) the other replica moves b under a. Naïvely replaying one replica's updates at the other produces an $a-b$ cycle disconnected from the root.

This is a widespread issue; indeed, many replicated file systems have serious anomalies, including incorrect or diverged states [2, Section 6 for some examples], violating the tree invariant [1]. However, concurrent moves are relatively rare in these systems[1] and it is important that we design a solution that has minimal overhead.

Solutions in the literature include non-atomic moves [2] (resulting in duplicate copies), reintroducing coordination [3] (the first to acquire lock will proceed; the other aborts), or requiring roll-backs [4] (the move operation ordered first proceeds, and all concurrent operations are rolled back). Najafzadeh et al. [3] shows that there can be no coordination-free solution to this problem that is not somehow anomalous.

To support low latency, high availability and safety, this paper introduces a new light-weight, coordination-free, safe, replicated CRDT [5] tree data structure, called *Maram*. Maram supports the usual operations to query the state, to add or to remove a node, and also supports an atomic *move* operation. The price to pay is that some move operations "lose", i.e., have no effect; achieving the same end result as previous correct approaches but at a lower cost.

Query and add are unremarkable. Remove marks the corresponding node as a "tombstone," but leaves it in the data structure, as is common in replicated data structures [6]. We show that moves can be divided into two cases: two concurrent *up-move*s are always safe. We devise a deterministic arbitration rule for conflicts of *down-move*: against a concurrent up-move, the up-move wins, and the down-move loses; against a concurrent down-move, the down-move with the highest priority (as defined in Section 4.2) wins and the other loses.

We prove Maram to be safe, even in the presence of concurrent updates (including moves), despite being coordination-free and without any roll-backs. Using the Why3 proof assistant, we apply the CISE proof methodology [7], with the following steps:

[1] For example, a file system trace we analyzed contained 1198823 operations in total, 20883 create operations, 49509 remove operations and just 547 move operations (70939 structural operations altogether).

1. *Sequential safety:* We show that the initial state satisfies the tree invariant, and that every update operation has a precondition strong enough to maintain the tree invariant.
2. *Convergence:* We show that any two operations that may execute concurrently commute.
3. *Precondition stability:* We show that for any two operations u, v that may execute concurrently, u preserves the precondition of v, and vice-versa.

Consequently, every state reachable from the initial state, sequentially or concurrently, satisfies the tree invariant.[2]

Maram satisfies an additional desirable property, *monotonic reads* [8]. This requires that a replica that has delivered some update will not roll it back.

This paper presents the principles of Maram, proves its correctness, and compares the performance of Maram to competing solutions in a simulated geo-replicated environment. The response time of Maram is the same as a naïve, uncoordinated design, and up to 15 times faster than (safe) lock-based designs. Furthermore, Maram stabilises (updates become definitive) three orders of magnitude faster than a safe rollback-based design.

This paper proceeds as follows. Section 2 formalises our system model, explains our proof methodology, and defines the tree invariant. In Section 3 we discuss the sequential correctness of a replicated tree. Section 4 proceeds with the proof of convergence and precondition stability, resulting in concurrent safety. In Section 5 we compare the performance of Maram with competing designs. Section 6 overviews the related literature. Finally, in Section 7 we discuss lessons learned and their significance.

2 Preliminaries

In this section we describe our system model and give a formal definition of the desired properties.

2.1 System Model

A system is a set of processes, distributed over a (high-latency, failure-prone) communication network. The processes have disjoint memory and processing capabilities, and they communicate through message passing.

2.1.1 State and invariant

The shared tree is *replicated* at a number of processes, called its *replicas*. The information managed by a replica on behalf of the data structure is called its *local state*.

The tree data structure is associated with an *invariant*, a predicate that must always be satisfied in the local state of a replica. Although evaluated locally, an invariant describes a global property, in the sense that it must be true at all replicas.

2.1.2 Operations

An unspecified client application submits an operation at some replica of its choice, which we call the *origin* replica of that operation. For availability, the origin replica carries out the operation without waiting to coordinate with other replicas.

The specification of an update operation comprises a *precondition* that indicates the domain of the operation and a *postcondition* that specifies the state after the operation executes. As

[2] We furthermore claim (without proof) that Maram is live, in the sense that, if every message sent is eventually delivered to some replica r_1, then, given some update originating at a replica r_2, its postcondition eventually takes effect at replica r_1.

discussed in more detail later, when the operation executes with no concurrency, its precondition guarantees that the operation terminates with the postcondition satisfied.

2.1.3 Updates

When a client submits an operation, the origin replica generates an *effector* (a side-effecting lambda), atomically applies the effector to the origin state, and sends the effector to all the other replicas. Every replica eventually receives and delivers the effector, atomically applying it to its own local state.[3]

We assume that effectors are delivered in causal order. This means that, if some replica that observed an effector u later generates an effector v, then any replica that observes v has previously observed u.[4]

In what follows, we ignore queries, and identify an update operation with executing its effector at all replicas.

2.2 Properties and associated proof rules

Consider some data structure (in this case, Maram) characterised by a safety *invariant*. We say that a state is *local-safe* if it satisfies the data structure's invariant. An update is *op-safe* if, starting from a local-safe state, it leaves it a local-safe state. The data structure is *safe* if every update is op-safe. According to the CISE logic [7], a distributed data structure is safe if 1. it is safe in sequential execution, 2. converges 3. and the precondition of each operation is stable under the effect of any other concurrent operation. We now detail these conditions.

2.2.1 Sequential safety

Consider an environment restricted to sequential execution (there is no concurrency). If the initial state is local-safe at every replica, and each update is op-safe, it follows that the data structure is safe under sequential execution. Classically, sequential op-safety implies that each operation's precondition satisfies the weakest-precondition of the invariant with respect to the operation [9]. Let us refine the proof obligations of this sequential safety step, i.e., local-safety under sequential execution.

The set of reachable states comprises the initial state, and all states transitively reachable as a result of executing updates sequentially. Formally, we note the set of reachable states Σ, a state σ, the initial state σ_{init}, an update u, the precondition of update u, Pre_u, and the set of updates U. When execution is sequential:

$$\sigma_{init} \in \Sigma \tag{1}$$

$$\forall u \in U, \sigma \in \Sigma \centerdot \sigma \models Pre_u \implies u(\sigma) \in \Sigma \tag{2}$$

where $\models$ is read *satisfies*. Σ is the smallest set satisfying (1) and (2) through a sequence of legal updates from the initial state. If *Inv* denotes the invariant, then we want

$$\forall \sigma \in \Sigma \centerdot \sigma \models Inv \tag{3}$$

Classically, if the initial state is safe and all sequential updates preserve the invariant, by induction, the data structure is sequentially safe. Formally,

$$\sigma_{init} \models Inv \tag{4}$$

$$\forall u \in U, \sigma, \sigma' \in \Sigma \centerdot \sigma \models Pre_u \wedge u(\sigma) = \sigma' \implies \sigma' \models Inv \tag{5}$$

[3] Since at this point the system is committed to this operation, the operation's precondition must be satisfied at the remote replica.

[4] In Section 7 we consider relaxing this requirement to eventual consistency, which states only that all updates are eventually delivered at all replicas.

2.2.2 Convergence

Let us now turn to concurrent execution, and consider the proof obligations for convergence.

If a replica initiates update u, while concurrently another replica initiates v, the first replica executes them in the order $u; v$ and the second one in the order $v; u$. To prevent divergence, the Strong Eventual Consistency (SEC) property [5] requires that any two replicas that delivered the same updates are in equivalent states. To satisfy SEC, effector functions are designed to commute, i.e., both orders above leave the data in the same state. We define commutativity as follows:

$$\forall u_1, u_2 \in U, \sigma, \sigma_1, \sigma_2 \in \Sigma \centerdot u_1(\sigma) = \sigma_1 \wedge u_2(\sigma) = \sigma_2 \implies u_2(\sigma_1) = u_1(\sigma_2) \tag{6}$$

2.2.3 Precondition stability

The final proof obligation for concurrent execution, is that the precondition of any effector is stable against (i.e., not negated by) an effector that may execute concurrently [7]: consider two updates u and v; if the execution of u does not make the precondition of v false, nor vice-versa (*precondition stability*), then executing u and v concurrently is op-safe. This must be true for all concurrent pairs of operations. Formally,

$$\forall u_1, u_2 \in U, \sigma, \sigma' \in \Sigma \centerdot \sigma \models (Inv \wedge Pre_{u_1} \wedge Pre_{u_2}) \wedge u_1(\sigma) = \sigma' \implies \sigma' \models Pre_{u_2} \tag{7}$$

This so-called CISE rule is a variant of rely-guarantee reasoning, adapted to a replicated system where effectors execute atomically.

2.2.4 Mechanized verification

In order to mechanically discharge the proof obligations listed above, we use Why3 system [10], augmented with the CISE3 plug-in [11]. Why3 is a framework used for the deductive verification of programs. The CISE3 plug-in automates the three proof rules described above, and generates the required sequential-safety, commutativity and stability checks. Why3 then computes a set of proof obligations, that are discharged via external theorem provers.

3 Sequential specification of a tree

The specification of a data structure consists of its state, a set of operations, and an invariant. In this section, we will develop a sequentially-safe specification of a tree.

3.1 State

The state of a tree data structure consists of a set of nodes, *Nodes*, and a relation from a child node to its parent, indicated by $\rightarrow$. The ancestor relation, $\rightarrow^*$ is defined as

$$\forall a, n \in Nodes \centerdot n \rightarrow^* a \implies n \rightarrow a \vee \exists p \in Nodes \centerdot n \rightarrow p \wedge p \rightarrow^* a \tag{8}$$

At initialization, the set of nodes consists of a single *root* node. The parent of the root is root itself. The initial state of the tree is thus $Nodes = \{root\}$ where $root \rightarrow root$.

A crucial aspect of the abstract representation of the tree is how to express the relation between nodes. Three choices are possible, either maintain a child-to-parent relation, a parent-to-child relation, or both. In particular, when implementing a tree, traversal efficiency depends on keeping both up and down pointers [12]. Considering that child-to-parent and parent-to-child relations describe a dual view of a tree (i.e., node p is the parent of node n iff node n is a descendent of node p) we selected the one that leads to a simpler specification. An advantage of using a child-to-parent relation is that it can be maintained as a function, as the tree properties ensure that each node has a unique parent. The alternative parent-to-child relation would require a more complex representation, e.g. a function that maps each node to its set of direct descendants, which would impact both the simplicity of the specification and the proof effort.

3.2 Invariant

The invariant of the tree data structure is as follows:

$$\begin{aligned} & root \in Nodes \land root \to root \land \\ & \forall \in Nodes \,.\, n \neq root \implies root \not\to n \end{aligned} \qquad (Root)$$

$$\land\ \forall n \in Nodes \,.\, \exists p \in Nodes \,.\, n \to p \qquad (Parent)$$

$$\land\ \forall n, p, p' \in Nodes \,.\, n \to p \land n \to p' \implies p = p' \qquad (Unique)$$

$$\land\ \forall n \in Nodes \,.\, n \to^* root \qquad (Reachable)$$

$$Inv \triangleq Root \land Parent \land Unique \land Reachable \qquad (9)$$

Clause *Root* states that the root node is present in *Nodes*, and is the only node to be its own parent. Clause *Parent* asserts that every node in the tree has a parent in the tree. Clause *Unique* requires the parent of a node to be unique. Clause *Reachable* imposes that the root is an ancestor of all nodes. We call this conjunction, Equation (9), the *tree invariant*.

A further invariant which forbids cycles (no node is an ancestor of itself, except root), can be derived:

$$\forall n \in Nodes \,.\, n \neq root \implies n \not\to^* n \qquad (Acyclic)$$

Since the parent relation inductively defines the ancestor relation, by *Unique* there is a unique path to a given ancestor of a node. By *Reachable*, the root node is an ancestor of every node in the tree. In this scenario, a cycle would require a node to have multiple parents, which is prevented by *Unique*.

3.3 Operations

We consider the following three structural operations on a tree: add, remove and move.

Add An add operation has two arguments: the node to be added, n, and its prospective parent, p. The add effector adds node n to *Nodes* and the mapping $n \to p$ to the parent relation. The postcondition of the add effector indicates this:[5]

$$Post_{add(n,p)} \triangleq n \in Nodes \land n \to p \qquad (10)$$

To ensure the tree invariant, we derive the precondition that n is a new node and p is already in the tree, i.e.,

$$Pre_{add(n,p)} \triangleq n \notin Nodes \land p \in Nodes \qquad (11)$$

[5] For readability, we simplify the postcondition to express only the changes caused by the operation. The part of the state not mentioned remains unaffected.

Precondition	Invariant clause			
	Root	*Parent*	*Unique*	*Reachable*
$add(n, p)$	$n \notin Nodes$	$p \in Nodes$	$n \notin Nodes$	$p \in Nodes$
$rem(n)$	$n \neq root$	$\forall n' \in Nodes \,.\, n' \not\rightarrow n$	`true`	$\forall n' \in Nodes \,.\, n' \not\rightarrow n$
$move(n, p')$	$n \neq root$	$p' \in Nodes$	`true`	$p' \in Nodes \wedge p' \neq n \wedge p' \not\rightarrow^* n$

Table 1: Precondition required by each operation to uphold specific clauses of the invariant

Let us see how this precondition is derived. If the add operation is updating a safe state, i.e., the starting state respects the invariant, and if the precondition is satisfied, then the update should maintain the invariant. Hereafter, we highlight the precondition clauses needed to ensure each part of the invariant.[6]

$$\frac{Inv \wedge \quad n \notin Nodes \qquad [\![add(n,p)]\!]}{Post_{add(n,p)} \wedge Root} \qquad \frac{Inv \wedge \quad p \in Nodes \qquad [\![add(n,p)]\!]}{Post_{add(n,p)} \wedge Parent}$$

$$\frac{Inv \wedge \quad n \notin Nodes \qquad [\![add(n,p)]\!]}{Post_{add(n,p)} \wedge Unique} \qquad \frac{Inv \wedge \quad p \in Nodes \qquad [\![add(n,p)]\!]}{Post_{add(n,p)} \wedge Reachable}$$

Table 1 lists the preconditions required by operations to preserve each invariant clause. With the derived preconditions, the add operation can be specified as follows:

$$\frac{\text{(Add-Operation)} \quad Inv \wedge n \notin Nodes \wedge p \in Nodes \qquad [\![add(n,p)]\!]}{Inv \wedge n \in Nodes \wedge n \rightarrow p}$$

If the add operation is issued on a state that is safe and contains p and not n, then n is added to the tree with parent p.

Remove operation Remove receives as argument a node n to be deleted. Its effector removes node n from the set of nodes. The postcondition of the remove operation indicates this effect:

$$Post_{rem(n)} \triangleq n \notin Nodes \tag{12}$$

Similarly to add, we list the predicates needed to preserve each clause of the invariant in Table 1. The remove operation can be specified as follows:

$$\frac{\text{(Remove-Operation)} \quad Inv \wedge n \neq root \wedge \forall\, n' \in Nodes \,.\, n' \not\rightarrow n \qquad [\![rem(n)]\!]}{Inv \wedge n \notin Nodes}$$

If a remove operation is issued on a safe state where n is not *root* and has no children, then n is removed from the tree.

Move operation The move operation takes two arguments: the node to be moved n, and the new parent p'. Its effector changes the parent of node n to p' as follows:

$$Post_{move(n,p')} \triangleq n \rightarrow p' \tag{13}$$

[6] Denoted in inference style, as in [13]. The condition above the line represents the pre-state, an update event is noted $[\![.]\!]$, and the condition below the line indicates the post-state.

To preserve the expected behaviour we require that the node to be moved is already present in the tree. We derive the safety clauses as shown in Table 1. Formally, the move operation can be specified as follows:

$$\frac{\begin{array}{l}(\textsc{Move-Operation})\\ Inv \wedge n \in Nodes \wedge n \neq root\\ \wedge\, p' \in Nodes \wedge p' \neq n \wedge p' \not\rightarrow^* n \qquad [\![move(n,p')]\!]\end{array}}{Inv \wedge n \rightarrow p'}$$

For the move operation to be safe, n is not the root, p' must be in the tree, n and p' are different, and p' is not a descendant of n. These last two conditions are needed to prevent move from creating a cycle of unreachable nodes, as we show with the following counterexample.

Consider a tree composed of nodes a and b. Root node R is the parent of node a, i.e., $a \rightarrow R$ and node a is the parent of node b, $b \rightarrow a$, and hence R is the ancestor of b, $b \rightarrow^* R$. Moving node a under node b will make both a and b unreachable from the root, and also form a cycle. This violates the invariant by invalidating the tree structure. To avoid this scenario, a precondition is needed that prevents moving a node underneath itself. When moving node n from its current parent to the new parent p', p' should not be a descendant of n, $p' \not\rightarrow^* n$.

3.4 Mechanized verification of the sequential specification

The mechanical proof, using Why3, of the above sequential specification requires some extra definitions and axioms.

To define reachability, we first define a path; a path is a sequence of nodes related by the parent relation. We denote the set of possible sequences of nodes[7] by S. The predicate determines the validity conditions for a path s between nodes x and y in state σ. If $x = y$, the path has length zero. Otherwise, the length of the path is greater than zero, where the first path element must be x, all contiguous path elements are related by the parent relation, and node y is the parent of the last path element. We say y is reachable from x if there exists a path from x to y. Formally,

$$\begin{aligned} path(\sigma, x, y, s) \triangleq\ & length(s) = 0 \wedge x = y \\ & \vee\, (length(s) > 0 \wedge s[0] = x \wedge \\ & s[length(s) - 1] \rightarrow y \wedge \\ & \forall\, 0 \leq i < length(s) - 1 \,.\, s[i] \rightarrow s[i+1]) \end{aligned} \tag{14}$$

$$reachability(\sigma, x, y) \triangleq \exists s \in S \,.\, path(\sigma, x, y, s) \tag{15}$$

To formalize the *path* predicate, we define a set of axioms as shown in Table 2. Axiom *path_to_parent* defines the singleton path of a node to its parent. The recursive composition of paths is axiomatized in *path_composition*. The transitivity property is defined in *path_transitivity*. Axiom *path_uniqueness* asserts there is a single path between two nodes. The *path_exclusion* expresses the conditions for excluding nodes from a path. Lastly, *path_separation* defines a convergence criterion essential for Why3's SMT solvers, asserting that the direction of the path is converging towards the root. Note that *path* and *rank* axioms are defined for non-root nodes because the operations' preconditions preclude applying them to the root.

We also require extra axioms to express the properties of the unaffected nodes in the case of add and move operations as shown in Table 3. The state σ_{add} is obtained by applying $add(n, p)$ operation on σ. The axiom *remaining_nodes_add* asserts that the paths already present in the tree remain in the tree after executing the add operation. Given that move operation updates σ to

[7] We use $s[n]$ to indicate the nth element in the sequence s.

Property name	Definition
path_to_parent	$\forall \sigma \in \Sigma \,.\, \forall x, y \in Nodes \,.\, x \rightarrow y \implies \exists s \in S \,.\, path(\sigma, x, y, s) \wedge s = [x]$
path_composition	$\forall \sigma \in \Sigma \,.\, \forall x, y, z \in Nodes \,.\, \exists s_1 \in S \,.\, path(\sigma, x, y, s_1) \wedge y \rightarrow z \implies \exists s_2 \in S \,.\, path(\sigma, x, z, s_2) \wedge s_2 = s_1 + [y]$
path_transitivity	$\forall \sigma \in \Sigma \,.\, \forall x, y, z \in Nodes, s_1, s_2 \in S \,.\, path(\sigma, x, y, s_1) \wedge path(\sigma, y, z, s_2) \implies \exists s_3 \in S \,.\, path(\sigma, x, z, s_3) \wedge s_3 = s_1 + s_2$
path_uniqueness	$\forall \sigma \in \Sigma \,.\, \forall x, y \in Nodes, s_1, s_2 \in S \,.\, path(\sigma, x, y, s_1) \wedge path(\sigma, x, y, s_2) \implies s_1 = s_2$
path_exclusion	$\forall \sigma \in \Sigma \,.\, \forall x, y, z \in Nodes, s \in S \,.\, x \not\rightarrow^* y \wedge path(\sigma, z, y, s) \implies x \notin s$
path_separation	$\forall \sigma \in \Sigma \,.\, \forall x, y, z \in Nodes, s_1, s_2 \in S \,.\, path(\sigma, x, y, s_1) \wedge path(\sigma, y, z, s_2) \wedge x \neq y \wedge x \neq z \wedge y \neq z \implies s_1 \cap s_2 = \emptyset$

Table 2: Properties of *path* predicate

Property name	Definition
remaining_nodes_add	$\forall \sigma \in \Sigma \,.\, \forall n' \in Nodes, s_1, s_2 \in seq(Nodes) \,.\, n' \neq n \wedge path(\sigma, n', root, s_1) \wedge path(\sigma_{add}, n', root, s_2) \implies s_1 = s_2$
descendants_move	$\forall \sigma \in \Sigma \,.\, \forall n' \in Nodes, s_1, s_2 \,.\, path(\sigma, n', c, s_1) \wedge path(\sigma_{move}, n', c, s_2) \implies s_1 = s_2$
remaining_nodes_move	$\sigma \in \Sigma \,.\, \forall n' \in Nodes, s_1, s_2 \in seq(Nodes) \,.\, n' \not\rightarrow^* n \wedge path(\sigma, n', root, s_1) \wedge path(\sigma_{move}, n', root, s_2) \implies s_1 = s_2$

Table 3: Properties of unaffected nodes for *add* and *move* operations, $\sigma_{add} = add(n, p)(\sigma)$ and $\sigma_{move} = move(n, p)(\sigma)$

σ_{move}, axiom *descendants_move* asserts that the descendants of the node being moved continue to be its descendants, and *remaining_nodes_move* asserts that other paths are not affected. These axioms are defined to ensure that the paths to the root, from nodes unaffected by move or add operations, remain unchanged. The specification proven using Why3 is available at [15].

4 Concurrent tree specification

In this section, we discuss the concurrent safety and convergence of the tree. In a sequential execution environment, as seen in Section 3, if the initial state and each individual update are safe, then all reachable states are safe. This is not true when executing concurrently on multiple replicas. In this case, as explained in Section 2.2, there are two extra proof obligations: ensuring that different replicas converge, despite effectors being executed concurrently in different orders, and ensuring that safety of an update is not violated by a concurrent update. First we discuss concurrent safety; convergence is deferred to Section 4.3, since the conflicts occurring in the latter can be addressed using the policies discussed in the former.

4.1 Precondition stability

We use the precondition stability rule of CISE logic (Section 2.2) to analyze the concurrent safety of our tree data structure. For each operation, we analyze whether it negates the precondition of any other concurrent operation. Formally, operation op_1 is stable under operation op_2 if,

$$\frac{Inv \wedge Pre_{op_1} \wedge Pre_{op_2} \qquad [\![op_2]\!]}{Inv \wedge Post_{op_2} \wedge Pre_{op_1}} \tag{16}$$

We first check for stability in the sequential specification. If this fails, then it is necessary to correct the specification, so that it does satisfy stability.

4.1.1 Stability of add operation

Concurrent adds Let us check the stability of the precondition of add against itself. Consider two operations $add(n_1, p_1)$ and $add(n_2, p_2)$. Using Equation (16),

$$\begin{aligned} Pre_{add(n_1,p_1)} &\triangleq n_1 \notin Nodes \wedge p_1 \in Nodes \\ Pre_{add(n_2,p_2)} &\triangleq n_2 \notin Nodes \wedge p_2 \in Nodes \\ Post_{add(n_2,p_2)} &\triangleq n_2 \in Nodes \wedge n_2 \rightarrow p_2 \end{aligned}$$

$$\frac{\begin{array}{l} Inv \quad \wedge \quad Pre_{add(n_1,p_1)} \quad \wedge \\ Pre_{add(n_2,p_2)} \wedge \; n_1 \neq n_2 \qquad [\![add(n_2,p_2)]\!] \end{array}}{Inv \wedge Post_{add(n_2,p_2)} \wedge Pre_{add(n_1,p_1)}} \tag{17}$$

The highlighted clause $n_1 \neq n_2$ is required for the stability condition. Indeed, the sequential specification does not disallow adding the same node at different replicas, and the clause $n \notin Nodes$ is unstable. Thus the analysis highlights a subtle error.

Concurrent remove Let us check the stability of the precondition of $add(n_1, p_1)$ against a concurrent $rem(n_2)$. Using (16), we get:

$$\begin{aligned} Pre_{add(n_1,p_1)} &\triangleq n_1 \notin Nodes \wedge p_1 \in Nodes \\ Pre_{rem(n_2)} &\triangleq n_2 \neq root \wedge \forall\, n' \in Nodes \,.\, n' \not\rightarrow n_2 \\ Post_{rem(n_2)} &\triangleq n_2 \notin Nodes \end{aligned}$$

Property name	Definition
skipping_abstraction	$\forall n \in Nodes_{con} \bullet n \notin TS \wedge \nexists n' \in Nodes_{con} \bullet n' \in TS \wedge n \rightarrow^* n' \iff n \in Nodes_{abs}$
keeping_abstraction	$\forall n \in Nodes_{con} \bullet n \notin TS \vee \exists n' \in Nodes_{con} \bullet n' \notin TS \wedge n' \rightarrow^* n \iff n \in Nodes_{abs}$

Table 4: Abstraction functions, $Nodes_{con}$ and $Nodes_{abs}$ denote the set of nodes in the concrete and abstract state respectively

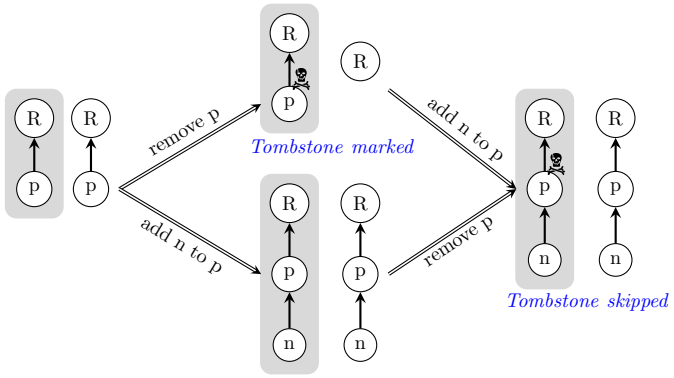


Figure 1: Resolving conflict of concurrent remove and add

$$\frac{Inv \wedge Pre_{add(n_1,p_1)} \wedge Pre_{rem(n_2)} \wedge n_2 \neq p_1 \quad [\![rem(n_2)]\!]}{Inv \wedge Post_{rem(n_2)} \wedge Pre_{add(n_1,p_1)}} \tag{18}$$

In the sequential specification, clause $p_1 \in Nodes$ in the precondition of add is unstable against a remove of its parent; performing those operations concurrently would be unsafe.

To fix this, we see two possible approaches. The classical way is to strengthen the precondition with coordination, for instance locking or using CAS to avoid concurrency. We reject this, as it conflicts with our objective of availability under partition. Our alternative is to weaken the specification thanks to coordination-free conflict resolution. We apply a common approach, which is to mark a node as deleted, as a so-called *tombstone*, without actually removing it from the data structure.[8]

We now distinguish a *concrete* state and its *abstract* view. We modify the specification to include a set of tombstones, *TS* (initially empty), in the concrete state. The abstract state is the resolved state as seen by some application using Maram. Depending on the requirements, the *abstraction function*, can either skip the tombstoned node including its descendants (the set of nodes that has the tombstoned node as an ancestor) or keep the tombstoned node in the presence of non-tombstoned descendants; we call them *skipping_abstraction* and *keeping_abstraction* respectively, with definitions as shown in Table 4.

To illustrate tombstones, consider the tree consisting of the root and a single child, as shown in Figure 1. Let us assume that the application chooses *keeping_abstraction*. One replica performs a remove of node p, while concurrently another replica adds n under p. In the first replica, node

[8] Ideally, one will remove the tombstone at some safe time in the future; this is non-trivial [14] and out of the scope of this paper.

p is marked as a tombstone in the concrete state (the shaded box). Thus, the abstract state shows node p removed. When the replicas exchange their updates, they converge to the state shown in the right-hand side of Figure 1. In the concrete state, node p is marked as a tombstone; however, since its descendant n is not a tombstone, now p is "revived" in the abstract view.

If the application chooses *skipping_abstraction*, the final abstract state will contain only the root node, skipping the tombstoned node and it's non-tombstoned children.

Accordingly, let us update the postcondition for remove:

$$Post_{rem(n)} \triangleq n \in TS \tag{19}$$

Let us now derive the predicates needed to preserve each clause of the invariant in this refined case.

$$\frac{Inv \wedge n \neq root \qquad [\![rem(n)]\!]}{Post_{rem(n)} \wedge Root} \qquad \frac{Inv \wedge \texttt{true} \qquad [\![rem(n)]\!]}{Post_{rem(n)} \wedge Parent}$$

$$\frac{Inv \wedge \texttt{true} \qquad [\![rem(n)]\!]}{Post_{rem(n)} \wedge Unique} \qquad \frac{Inv \wedge \texttt{true} \qquad [\![rem(n)]\!]}{Post_{rem(n)} \wedge Reachable}$$

To maintain sequential safety in the modified remove specification, the precondition forbids removing the root node. As the remove operation doesn't alter the tree structure, reachability is not impacted. The refined specification of the remove operation is as follows:

$$\frac{\text{(Remove-Operation)}}{\dfrac{Inv \wedge n \neq root \qquad [\![rem(n)]\!]}{Inv \wedge n \in TS}}$$

Concurrent move Next we check the stability of the precondition of add under a concurrent move operation. Let us consider two operations $add(n_1, p_1)$ and $move(n_2, p_2')$. Using (16), we get

$$\begin{aligned} Pre_{add(n_1,p_1)} &\triangleq n_1 \notin Nodes \wedge p_1 \in Nodes \\ Pre_{move(n_2,p_2')} &\triangleq n_2 \in Nodes \wedge n_2 \neq root \wedge p_2' \in Nodes \\ &\quad \wedge p_2' \neq n_2 \wedge p_2' \not\rightarrow^* n_2 \\ Post_{move(n_2,p_2')} &\triangleq n_2 \rightarrow p_2' \end{aligned}$$

$$\frac{\begin{array}{l} Inv \ \wedge \ Pre_{add(n_1,p_1)} \ \wedge \\ Pre_{move(n_2,p_2')} \wedge \texttt{true} \end{array} \qquad [\![move(n_2, p_2')]\!]}{Inv \wedge Post_{move(n_2,p_2')} \wedge Pre_{add(n_1,p_1)}} \tag{20}$$

We see that the precondition of add is stable with a concurrent move.

4.1.2 Stability of remove operation

Concurrent add Consider the sequential specification of two operations $rem(n_1)$ and $add(n_2, p_2)$. Using (16), we get

$$\begin{aligned} Pre_{rem(n_1)} &\triangleq n_1 \neq root \wedge \forall\, n' \in Nodes \,.\, n' \not\rightarrow n_1 \\ Pre_{add(n_2,p_2)} &\triangleq n_2 \notin Nodes \wedge p_2 \in Nodes \\ Post_{add(n_2,p_2)} &\triangleq n_2 \in Nodes \wedge n_2 \rightarrow p_2 \end{aligned}$$

$$\frac{Inv \wedge Pre_{rem(n_1)} \wedge Pre_{add(n_2,p_2)} \wedge \boxed{n_1 \neq p_2} \qquad [\![add(n_2,p_2)]\!]}{Inv \wedge Post_{add(n_2,p_2)} \wedge Pre_{rem(n_1)}} \tag{21}$$

We see that the clause that node n_1 has to be a leaf node is not satisfied if $n_1 = p_2$ since add operation introduces a child node under p_2. However, the refined specification with tombstones as described above doesn't require the node n_1 to be a leaf node. So that solution fixes this conflict as well.

Concurrent remove Consider the sequential specification of two remove operations $rem(n_1)$ and $rem(n_2)$. Using (16), we get

$$\begin{aligned} Pre_{rem(n_1)} &\triangleq n_1 \neq root \wedge \forall\, n' \in Nodes \,.\, n' \not\rightarrow n_1 \\ Pre_{rem(n_2)} &\triangleq n_2 \neq root \wedge \forall\, n' \in Nodes \,.\, n' \not\rightarrow n_2 \\ Post_{rem(n_2)} &\triangleq n_2 \notin Nodes \end{aligned}$$

$$\frac{Inv \wedge Pre_{rem(n_1)} \wedge Pre_{rem(n_2)} \wedge \boxed{\texttt{true}} \qquad [\![rem(n_2)]\!]}{Inv \wedge Post_{rem(n_2)} \wedge Pre_{rem(n_1)}} \tag{22}$$

We see that the remove operation is stable under a concurrent remove. Furthermore, the refined specification is also stable since it adds n_1 and n_2 to TS.

Concurrent move Consider the sequential specification of two operations $rem(n_1)$ and $move(n_2, p_2')$. Using (16), we get

$$\begin{aligned} Pre_{rem(n_1)} &\triangleq n_1 \neq root \wedge \forall\, n' \in Nodes \,.\, n' \not\rightarrow n_1 \\ Pre_{move(n_2,p_2')} &\triangleq n_2 \in Nodes \wedge n_2 \neq root \wedge p_2' \in Nodes \\ &\quad \wedge p_2' \neq n_2 \wedge p_2' \not\rightarrow^* n_2 \\ Post_{move(n_2,p_2')} &\triangleq n_2 \rightarrow p_2' \end{aligned}$$

$$\frac{\begin{array}{l} Inv \quad \wedge \quad Pre_{rem(n_1)} \quad \wedge \\ Pre_{move(n_2,p_2')} \wedge \boxed{n_1 \neq p_2'} \end{array} \qquad [\![move(n_2,p_2')]\!]}{Inv \wedge Post_{move(n_2,p_2')} \wedge Pre_{rem(n_1)}} \tag{23}$$

We see that the clause for the remove operation that n_1 should be a leaf node is violated if a node is moved under it. Again, observe that the refined specification of remove mitigates this issue due to the absence of the violation-causing clause.

4.1.3 Stability of move operation

Concurrent add Consider the sequential specification of two operations $move(n_1, p_1')$ and $add(n_2, p_2)$. Using (16), we get

$$\begin{aligned} Pre_{move(n_1,p_1')} &\triangleq n_1 \in Nodes \wedge n_1 \neq root \wedge p_1' \in Nodes \\ &\quad \wedge p_1' \neq n_1 \wedge p_1' \not\rightarrow^* n_1 \\ Pre_{add(n_2,p_2)} &\triangleq n_2 \notin Nodes \wedge p_2 \in Nodes \\ Post_{add(n_2,p_2)} &\triangleq n_2 \in Nodes \wedge n_2 \rightarrow p_2 \end{aligned}$$

$$\frac{\begin{array}{l} Inv \wedge Pre_{move(n_1,p_1')} \wedge \\ Pre_{add(n_2,p_2)} \wedge \texttt{true} \end{array} \qquad [\![add(n_2,p_2)]\!]}{Inv \wedge Post_{add(n_2,p_2)} \wedge Pre_{move(n_1,p_1')}} \tag{24}$$

We see that the precondition of move is stable with a concurrent add operation.

Concurrent remove Consider the sequential specification of two remove operations $move(n_1, p_1')$ and $rem(n_2)$. Using (16), we get

$$\begin{aligned} Pre_{move(n_1,p_1')} &\triangleq n_1 \in Nodes \wedge n_1 \neq root \wedge p_1' \in Nodes \\ &\quad \wedge p_1' \neq n_1 \wedge p_1' \not\mapsto^* n_1 \\ Pre_{rem(n_2)} &\triangleq n_2 \neq root \wedge \forall\, n' \in Nodes \,.\, n' \not\mapsto n_2 \\ Post_{rem(n_2)} &\triangleq n_2 \notin Nodes \end{aligned}$$

$$\frac{\begin{array}{l} Inv \;\wedge\; Pre_{move(n_1,p_1')} \;\wedge \\ Pre_{rem(n_2)} \wedge n_2 \neq p_1' \end{array} \qquad [\![rem(n_2)]\!]}{\begin{array}{ll} Inv \wedge Post_{rem(n_2)} & \wedge Pre_{move(n_1,p_1')} \end{array}} \tag{25}$$

We observe here that removing n_2 violates the clause $p_1' \in Nodes$ if n_2 and p_1' are the same. However, in our refined specification, the postcondition of remove is $n_2 \in TS$, keeping the clause $p_1' \in Nodes$ stable.

Concurrent move Consider the sequential specification of two operations $move(n_1, p_1')$ and $move(n_2, p_2')$. Using (16), we get

$$\begin{aligned} Pre_{move(n_1,p_1')} &\triangleq n_1 \in Nodes \wedge n_1 \neq root \wedge p_1' \in Nodes \\ &\quad \wedge p_1' \neq n_1 \wedge p_1' \not\mapsto^* n_1 \\ Pre_{move(n_2,p_2')} &\triangleq n_2 \in Nodes \wedge n_2 \neq root \wedge p_2' \in Nodes \\ &\quad \wedge p_2' \neq n_2 \wedge p_2' \not\mapsto^* n_2 \\ Post_{move(n_2,p_2')} &\triangleq n_2 \to p_2' \end{aligned}$$

$$\frac{\begin{array}{l} Inv \quad \wedge \quad Pre_{move(n_1,p_1')} \quad \wedge \\ Pre_{move(n_2,p_2')} \wedge p_1' \not\mapsto^* n_2 \end{array} \qquad [\![move(n_2,p_2')]\!]}{Inv \wedge Post_{move(n_2,p_2')} \wedge Pre_{move(n_1,p_1')}} \tag{26}$$

We see here that a concurrent move of p_1 or it's ancestor invalidates the precondition clause $p_1' \not\mapsto^* n_1$ that prevents a cycle. We discuss this in more detail in Section 4.2 and explain how we refine the specification for stability.

Table 5 shows the summary of the stability analysis on the sequential specification discussed in Section 3. Symbol ✔ indicates that the precondition of the operation in that row is stable under the operation in the column. In case of instability, a condition replaces it, indicating the condition under which it is stable.

Stability	Operations		
	$add(n_2, p_2)$	$rem(n_2)$	$move(n_2, p_2')$
$add(n_1, p_1)$	$n_1 \neq n_2$	$p_1 \neq n_2$	✔
$rem(n_1)$	$n_1 \neq p_2$	✔	$n_1 \neq p_2'$
$move(n_1, p_1')$	✔	$p_1' \neq n_2$	$p_1' \not\to^* n_2$

Table 5: Stability analysis of sequential specification discussed in Section 3

Property name	Definition
critical_ancestors	$\{a \in Nodes \,.\, p' \to^* a \wedge n \not\to^* a\}$
critical_descendants	$\{d \in Nodes \,.\, d \to^* n\}$

Table 6: Critical ancestors and critical descendants of $move(n, p')$

4.2 Safety of concurrent moves

We closely examine how a move operation on a remote replica might affect the precondition of a concurrent move in the local replica. Consider an operation $move(n, p')$. In a sequential execution, precondition clause $p' \not\to^* n$ forbids moving a node under itself (which would cause a cycle). However a concurrent move of p' under n will not preserve the precondition of the operation, $p' \not\to^* n$, resulting in a cycle.

This issue generalizes to p' or its ancestor concurrently moving under n or a descendant of n. For easy reference, we call this move as a *cycle-causing-concurrent-move*. Observe that the precondition prevents an ancestor of n moving under itself in sequential execution. Therefore, only the ancestors of p' that are not ancestors of n would lead to a cycle. We call this set of ancestors *critical ancestors*, and the set of n and its descendants *critical descendants* as defined in Table 6.

Consider two concurrent move operations $move(n, p')$ and $move(p', n)$. Figure 2 shows the critical ancestors and critical descendants of both move operations. The node l is common ancestor of both n and p' farthest from the root. The critical ancestors and critical descendants of $move(n, p')$ are grouped together in the dark gray region with and without a border respectively, and that of $move(p', n)$ are grouped together in the light gray region.

Note that the set of critical descendants of a move overlaps with the critical ancestors set of its corresponding cycle-causing-concurrent-move. Hence, we consider only the critical ancestors of move operations.

Let us take a step back and analyze the types of move operations. Some move operations result in a node moving farther away from the root, called *down-move*s, and another set of move operations result in the node moving nearer to the root, or to remain at the same distance from the root, called *up-move*s. We define *rank* as the distance of a node from the root node, as follows:

$$rank(root) = 0 \tag{27}$$

$$rank(n) = rank(p) + 1 \mid \forall n, p \in Nodes \,.\, n \to p \tag{28}$$

$$up\text{-}move(n, p') \implies rank(n) > rank(p') \tag{29}$$

$$down\text{-}move(n, p') \implies rank(n) \leq rank(p') \tag{30}$$

Consider a move operation, $move(n, p')$, moving node n at the same level or towards the root, i.e., an up-move. This gives us that $rank(n) > rank(p')$. In this case, the rank of a critical

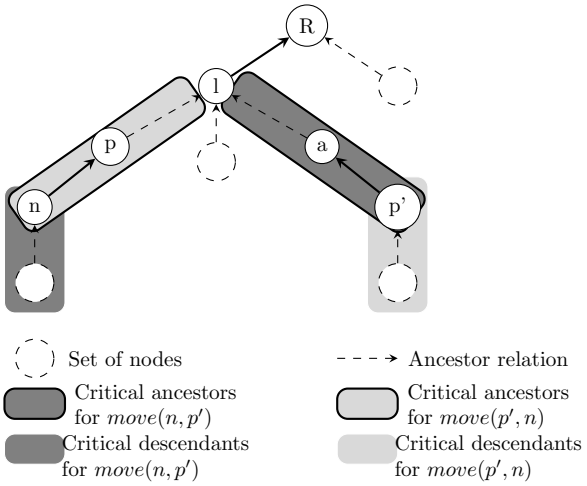


Figure 2: Critical ancestors and critical descendants

descendant will be always greater than the rank of a critical ancestor. Formally,

$$\begin{aligned} &\forall n, p, p', d, a \in \mathit{Nodes} \centerdot n \rightarrow p \wedge \mathit{rank}(n) > \mathit{rank}(p') \\ &\quad \wedge d \rightarrow^* n \wedge p' \rightarrow^* a \implies \mathit{rank}(d) > \mathit{rank}(a) \end{aligned} \tag{31}$$

This implies that a cycle-causing-concurrent-move can only be a down-move. Hence, we have that concurrent up-moves are safe; stability issues can occur only between two concurrent down-moves, or between an up-move and a down-move.

Our next step is to design a coordination-free conflict resolution policy for the moves that conflict. The conflict resolution policy is required if both the concurrent move operations move a node in the set of critical ancestors of the other. If we have up-moves, we apply the effect of the operation. In case of a concurrent down-move and up-move, up-move wins and the down-move is skipped. In case of concurrent down-moves, we apply a deterministic conflict resolution policy; the operation with highest *priority number* wins. The priority number of a move operation is specific to each application, with a condition that it must be unique for each move.

Contrast our approach with the alternative that uses shared-exclusive locks for concurrent moves [3]. Consider concurrent operations $move(n, p')$, moving node n under p', and $move(p', n)$, moving node p' under n. These operations compete for a lock. The one that succeeds first will apply its move, blocking the other. When it releases the lock, this releases the second one, but its precondition is no longer valid and it cannot execute. Thereby, safety is preserved, at the cost of aborting the second move. This work essentially achieves the same end result, but without the overhead of locking. Our experiments in Section 5 show the performance difference.

4.3 Convergence

As discussed in Section 2.2, to ensure convergence, we design the data structure such that concurrent updates commute [5]. Add and remove operations result in adding the added and removed node to *Nodes* and *TS* respectively. Since set union is commutative, each of these two operations commutes with itself and with the other.

The move operation changes the parent pointer of a node. It commutes with add and remove, since it doesn't have an effect on set membership.

Commutativity	Operations		
	$add(n_2, p_2)$	$rem(n_2)$	$move(n_2, p_2')$
$add(n_1, p_1)$	✔	✔	✔
$rem(n_1)$	✔	✔	✔
$move(n_1, p_1')$	✔	✔	$\neg(n_1 = n_2 \wedge p_1' \neq p_2')$

Table 7: Result of commutativity analysis of the sequential specification discussed in Section 3

However, observe that in the sequential specification two moves do not commute, if the same node is moved to two different places. This issue is fixed by the conflict resolution policy discussed earlier. The results of the commutativity analysis is show in Table 7.

4.4 Safe specification of a replicated tree

Incorporating the stability and commutativity analysis results and the design refinements, we have a coordination-free, safe and convergent replicated tree data structure as shown in Figure 3. The state now consists of a set of nodes, *Nodes*, and tombstones, *TS*. Since the tombstones also form part of the tree, they also have to maintain the tree structure. So the invariants talk about the set of nodes, that includes tombstones.

We also introduce some definitions to help define the coordination-free and conflict-free up-move and down-move operations. We define an operation as a tuple of the type (add, remove, up-move or down-move), the parameters, and a priority number. The priority number can be specific to the application using the data structure; the only condition being that each up-move operation or down-move operation should get a unique priority number. We define `C` as the list of operations executed concurrently with the operation in consideration. We also define operations on critical ancestors as *crit_anc_overlap*, where the node being moved is a member of the set of critical ancestors of the other operation.

With the help of these definitions, we define the effects of up-move and down-move operations in three parts: the actual precondition needed to ensure sequential safety, the conflict resolution condition (highlighted) and the update on the state.

4.5 Mechanized verification of the concurrent specification

We use the CISE3 plug-in, presented in Section 2.2.4, to identify conflicts as shown in Tables 5 and 7. Given the sequential specification from Section 3, CISE3 automatically generates a set of meta-operations to check stability and commutativity of executing pairs of operations.

4.5.1 Provable concurrent execution

We update the Why3 specification according to the conflict resolution policies from Section 4.4. For example, for the add operation we place the new precondition that nodes must be uniquely identified:

```
assume { ... ∧ n1 ≠ n2 }
```

Next, we refine the definition of type `state` to include tombstones, as follows:

```
type state = { mutable nodes: fset elt; ...;
      mutable tombstones: fset elt; }
```

State: $Nodes \times TS$
Invariant:

$$root \in Nodes \land root \to root \land root \notin TS$$
$$\land\ \forall n \in Nodes \,.\, n \neq root \implies root \not\to n \quad (Root)$$
$$\land\ \forall n \in Nodes \,.\, n \neq root \land \exists p \in Nodes \,.\, n \to p \quad (Parent)$$
$$\land\ \forall n, p, p' \in Nodes \,.\, n \to p \land n \to p' \implies p = p' \quad (Unique)$$
$$\land\ \forall n \in Nodes \,.\, n \neq root \implies n \to^* root \quad (Reachable)$$

(Add-Operation)

$$\frac{Inv \land p \in Nodes \land n \notin Nodes \qquad [\![add(n,p)]\!]}{Inv \land n \in Nodes \land n \to p}$$

(Remove-Operation)

$$\frac{Inv \land n \neq root \qquad [\![rem(n)]\!]}{Inv \land n \in TS}$$

$$operation \triangleq (type, params, priority)$$
$$\mathtt{C} \triangleq set\ of\ concurrent\ operations$$
$$crit\_anc\_overlap(op_1, op_2) \triangleq op_1.params.n \in critical\_ancestor(op_2)\ \land op_2.params.n \in critical\_ancestor(op_1)$$

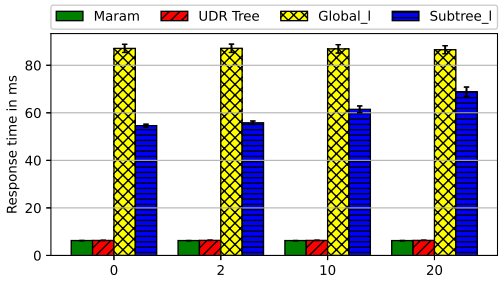
(Down-move-Operation)

$$\frac{\begin{array}{l} Inv \land n \in Nodes \land n \neq root \\ \land\ p' \in Nodes \land n \neq p' \land p' \not\to^* n \\ \nexists op \in \mathtt{C} \,.\, op.type = up\text{-}move \\ \quad \land\, op.params.n = n \land\ op.priority > priority \end{array} \qquad [\![up\text{-}move(n, p')]\!]}{Inv \land n \to p'}$$

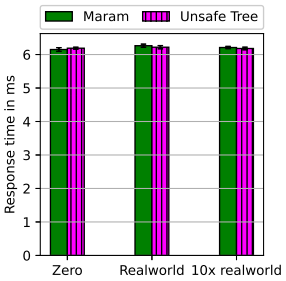
(Up-move-Operation)

$$\frac{\begin{array}{l} Inv \land n \in Nodes \\ \land n \neq root \land p' \in Nodes \\ \land n \neq p' \land p' \not\to^* n \\ \nexists op \in \mathtt{C} \,.\, op.type = up\text{-}move \\ \quad \land\, (crit\_anc\_overlap(down\text{-}move(n, p'), op) \lor\ op.params.n = n) \\ \land \nexists op \in \mathtt{C} \,.\, op.type = down\text{-}move \\ \quad \land\, (crit\_anc\_overlap(down\text{-}move(n, p'), op) \lor\ op.params.n = n) \\ \quad \land\, op.priority > priority \end{array} \qquad [\![down\text{-}move(n, p')]\!]}{Inv \land n \to p'}$$

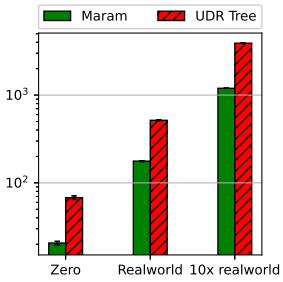
Figure 3: Concurrent specification of Maram



(a) Response time for different conflict rates



(b) Overhead of conflict resolution for different latencies



(c) Stabilization time for different latencies in logarithmic scale

Figure 4: Experimental results. Each bar is the average of 15 runs, the error bars show standard deviation

We update the specification of the `rem` accordingly:

```
val rem (n : elt) (s : state) : unit
  ensures  { s.tombstones = add n (old s).tombstones }
```

where `add` stands for the logical adding operation on sets.

Finally, the implementation of the conflict resolution policy for a pair of `move` operations requires us to be a bit more creative. We update the `state` type definition to include ranking and critical ancestors information. We implement a custom analysis `move_refined` operation since concurrent operations are not available off-the-shelf in Why3, a framework for verification of sequential specifications. We encode the arguments of two move operations as arguments of the `move_refined` operation: `n1` (`n2`), `np1` (`np2`), and `pr1` (`pr2`) stand for the node to be moved, the new parent, and the unique priority levels respectively, of the first (second) move.

All analysis functions, except `move_refined`, are automatically generated by the CISE3 plug-in of Why3. Finally, 55 verification conditions are generated for the implementation and given specification of `move_refined`. All of these are automatically verified, using a combination of SMT solvers. The specification and the proof results are available at [15].

Latency	Replicas		
	Paris	Bangalore	New York
Paris	0	144	75
Bangalore	144	0	215
New York	75	215	0

Table 8: Real world latency configurations in ms

5 Evaluation

This paper presents the design of a coordination-free, safe, convergent, and highly available replicated tree. The specification of Maram doesn't require any synchronization to execute an operation; this implies that the design is coordination-free. Sections 3 and 4 provide a mechanized proof that our design is safe and convergent. In this section, we conduct an evaluation to showcase the high availability of our design.

We measure availability in two parts - *response time* and *stabilization time*. The first metric, *response time*, is the time taken to log and acknowledge a client request. Recall that the effect of a move operation in our specification consists of either updating the state, or a skip. The effect of the update will be definitive only after being aware of all its concurrent operations. In order to measure this, we introduce a metric called *stabilization time*. Stabilization time measures the duration for which an update is in a transient state.

We run the experiments[9] with three replicas connected in a mesh with a FIFO connection and simulate different network latencies, zero latency, real world latency as shown in Table 8 and 10 times real world latency. Our warm-up workload, a mix of add, remove and move operations, creates a tree with 997 nodes including the root. We then have concurrent workloads[10] on the three replicas, varying conflict rates at 0%, 2%, 10%, and 20%.

We compare Maram with three solutions from the literature: (i) UDR tree (short for Undo-Do-Redo tree) [4]; (ii) all move operations acquiring a global lock (Global_l); and, (iii) move operations acquiring read locks on critical ancestors and write lock on the moving node (Subtree_l) [3].

The average response time for each design for different conflict rates with latency configuration 2 (Table 8) are shown in Figure 4a. Observe that Maram and UDR tree show the same average response time; owing to the synchronization-free design. The response time for Subtree_l [3] increases with an increase in the conflict rate due to lock contention, whereas that of Global_l is the same across all conflict rates since the proportion of lock-acquiring-moves remains the same.

Figure 4c shows the average stabilization time for our design and the UDR tree design [4] on a logarithmic scale, for different latency configurations. Our solution gives lower stabilization time, since only down-moves have transient state in the case of Maram whereas for a UDR tree [4] all operations are in transient state until a local replica asserts that there are no more concurrent operations.[11]

Next, we run an experiment to measure the overhead introduced by the conflict resolution policy. As a lower bound, we compare the response time of Maram with a naïve unsafe implementation, that uses a simple eventual consistency approach, and thus is not safe. Figure 4b shows

[9] On DELL PowerEdge R410 machine with 64 GB RAM, and 24 cores @2.40GHz Intel Xeon E5645 processor.

[10] 250 operations per replica - 60% add, 12% remove, 14% upmove and 14% downmove.

[11] Note here that Maram's stabilisation time does not depend on conflict rate, but only on the proportion of down-moves in the workload. As this proportion grows towards 100%, the stabilisation time of Maram tends to be the same as that of UDR.

Independent	Operations		
	$add(n_2, p_2)$	$rem(n_2)$	$move(n_2, p_2')$
$add(n_1, p_1)$	$p_1 \neq n_2$	✔	✔
$rem(n_1)$	$n_1 \neq n_2$	✔	✔
$move(n_1, p_1')$	$n_1 \neq n_2 \vee p_1' \neq n_2$	✔	$n_2 \notin descendants(n_1)$

Table 9: Result of dependency analysis

the response time of both the designs. For both the implementations, the tree is constructed in the critical path of the call from the logs. For Maram metadata is also computed at the same time. Since the cost of metadata computation is negligible compared to the cost of tree creation, Maram has negligible overhead.

6 Related work

Several works have addressed the problem of designing a replicated tree. Martin et al. [16] introduce some designs for conflict-free replicated tree data types. They use set CRDTs to construct replicated trees with different semantics. Add and remove operations are supported in their design. However they do not consider move operations.

Kleppmann et al. [4] propose the UDR tree, which supports atomic move operations, using the notion of opsets. Opsets totally order all operations eventually. This is more expensive than our solution based on partial order. When a new operation is performed, all the later operations are redone. Thus all operations pay a heavy price, and not just the conflicting moves. But UDR requires eventually consistent delivery layer, whereas Maram requires a more expensive causal delivery layer.

Najafzadeh et al. [3] designs the replicated tree called Subtree_l in Section 5. Their solution introduces coordination; acquires read locks on the critical ancestors, and a write lock on the node being moved. This approach is not available under partition, but only move operations pay an overhead.

Tao et al. [2] propose a replicated tree with a move operation that does not require any coordination between replicas, replacing each move with non-atomic copy and delete operations. This might lead to having multiple copies of the same node.

Compared to all the above solutions, our design supports atomic move operation that depends on partial ordering without acquiring any locks. An atomic update provides all or no guarantee, i.e., either the update is applied or it is not. Ensuring atomicity avoids partial execution of updates.

Kaki et al. [17] introduce the concept of Mergeable Replicated Data Types (MRDTs) inspired by three-way-merge. The safety of an MRDT binary tree depends on the labeling of the child-parent relations (whether it belongs to the right or left of the ancestor). It also requires to keep track of all the ancestor relations apart from the parent-child relations. A generic MRDT tree can be considered as an extension to the MRDT binary tree, but requires tracking all ancestor relations and a complex lexicographical ordering when concretizing the merged result.

7 Discussion

In this section we discuss the details that might effect the implementation of Maram data structure.

7.0.1 Moving from causal consistency to eventual consistency

Houshmand and Lesani [18] provide an analysis for relaxing the requirement of causal delivery to eventual consistency, called dependency analysis. In a nutshell, dependency analysis checks whether an operation is independent from the result of a previous operation. Table 9 shows the results of dependency analysis of Maram. Observe that no operations are dependent on remove, and add and remove are not dependent on move. All operations have conditional dependency with some other operation that warrants causal delivery. This analysis supports our choice of causal consistency, since there is no fully independent operation.

7.0.2 Message overhead for conflict resolution

Maram transmits the set of critical ancestors, priority number, and vector clocks[12]. The size of the set of critical ancestors depends on the depth of the subtree comprising the common ancestor (farthest from the root) of the node being moved and the destination parent. The size of the vector clock is dependent on the number of replicas (for n replicas, an integer array of size n) and priority number is a single number which is application-specific.

8 Conclusion

This paper presents the design of a light-weight, coordination-free, safe, convergent and highly available replicated tree data structure, Maram. We provide mechanized proof of safety and convergence of Maram, and experimentally demonstrate the efficiency of the design by comparing it with existing solutions.

References

[1] Bjørner, "Models and software model checking of a distributed file replication system," in *Formal Methods and Hybrid Real-Time Systems*, 2007.

[2] Tao et al., "Merging semantics for conflict updates in geo-distributed file systems," in *SYSTOR*, 2015.

[3] Najafzadeh et al., "Co-design and verification of an available file system," in *VMCAI*, 2018.

[4] Kleppmann et al., "OpSets: Sequential specifications for replicated datatypes," 2018. [Online]. Available: http://arxiv.org/abs/1805.04263

[5] Shapiro et al., "Conflict-free replicated data types," in *SSS*, 2011.

[6] Attiya et al., "Specification and complexity of collaborative text editing," in *PODC*, 2016.

[7] Gotsman et al., "'Cause I'm Strong Enough: Reasoning about consistency choices in distributed systems," in *POPL*, 2016.

[8] Terry et al., "Session guarantees for weakly consistent replicated data," in *PDIS*, 1994.

[9] Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *CACM*, vol. 18, no. 8, 1975.

[12] We assume that each replica works as a single-threaded process.

[10] Filliâtre and Paskevich, "Why3 – Where Programs Meet Provers," in *ESOP*, 2013.

[11] Meirim et al., "CISE3: Verifying weakly consistent applications with why3," 2020. [Online]. Available: http://arxiv.org/abs/2010.06622

[12] Tao, "Ensuring availability and managing consistency in geo-replicated file systems," PhD Thesis Sorbonne-Université–Université Pierre et Marie Curie, 2017.

[13] Kaki et al., "Safe replication through bounded concurrency verification," in *OOPSLA*, 2018.

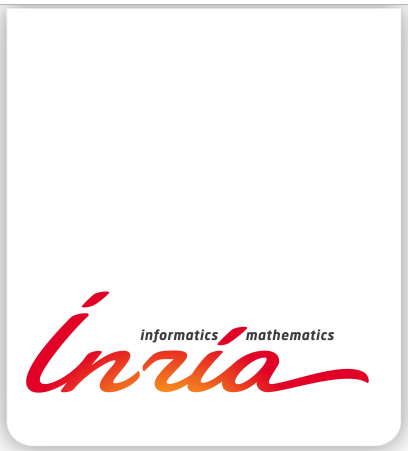
[14] Baquero et al., "Making operation-based CRDTs operation-based," in *DAIS*, 2014.

[15] Maram proof files, 2021. [Online]. Available: https://fmeirim.github.io/Maram_proofs/

[16] Martin et al., "Abstract unordered and ordered trees CRDT," INRIA, Tech. Rep. RR-7825, 2011.

[17] Kaki et al., "Mergeable replicated data types," in *OOPSLA*, 2019.

[18] Houshmand and Lesani, "Hamsaz: Replication coordination analysis and synthesis," in *POPL*, 2019.



RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399