Specification and Implementation of Replicated List: The Jupiter Protocol Revisited

Hengfeng Wei
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
hfwei@nju.edu.cn

Yu Huang[1]
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
yuhuang@nju.edu.cn

Jian Lu
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
lj@nju.edu.cn

Abstract

The replicated list object is frequently used to model the core functionality of replicated collaborative text editing systems. Since 1989, the convergence property has been a common specification of a replicated list object. Recently, Attiya et al. proposed the strong/weak list specification and conjectured that the well-known Jupiter protocol satisfies the weak list specification. The major obstacle to proving this conjecture is the mismatch between the global property on all replica states prescribed by the specification and the local view each replica maintains in Jupiter using data structures like 1D buffer or 2D state space. To address this issue, we propose CJupiter (Compact Jupiter) based on a novel data structure called n-ary ordered state space for a replicated client/server system with n clients. At a high level, CJupiter maintains only a single n-ary ordered state space which encompasses exactly all states of each replica. We prove that CJupiter and Jupiter are equivalent and that CJupiter satisfies the weak list specification, thus solving the conjecture above.

2012 ACM Subject Classification Computing methodologies → Distributed computing methodologies, Software and its engineering → Correctness, Human-centered computing → Collaborative and social computing systems and tools

Keywords and phrases Collaborative text editing systems, Replicated list, Concurrency control, Strong/weak list specification, Operational transformation, Jupiter protocol

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2018.12

Related Version A full version is available at [25], `https://arxiv.org/abs/1708.04754`.

Funding This work is supported by the National 973 Program of China (No. 2015CB352202) and the National Natural Science Foundation of China (No. 61690204, 61702253).

1 Introduction

Collaborative text editing systems, like Google Docs [2], Apache Wave [1], or wikis [11], allows multiple users to concurrently edit the same document. For availability, such systems often replicate the document at several *replicas*. For low latency, replicas are required to

[1] Contact Author.

respond to user operations immediately without any communication with others and updates are propagated asynchronously.

The *replicated list object* has been frequently used to model the core functionality (e.g., insertion and deletion) of replicated collaborative text editing systems [8, 13, 26, 5]. A common specification of a replicated list object is the *convergence* property, proposed by Ellis et al. [8]. It requires the *final lists at all replicas* be identical after executing the same set of user operations. Recently, Attiya et al. [5] proposed the strong/weak list specification. Beyond the convergence property, the strong/weak list specification specifies global properties on *intermediate states* going through by replicas. Attiya et al. [5] have proved that the existing RGA protocol [16] satisfies the strong list specification. Meanwhile, it is *conjectured* that the well-known Jupiter protocol [13, 26], which is behind Google Docs [3] and Apache Wave [4], satisfies the weak list specification.

Jupiter adopts a *centralized server* replica for propagating updates [2], and client replicas are connected to the server replica via FIFO channels; see Figure 1 [3]. Jupiter relies on the technique of operational transformations (OT) [8, 20] to achieve convergence. The basic idea of OT is for each replica to execute any local operation immediately and to transform a remote operation so that it takes into account the concurrent operations previously executed at the replica. Consider a replicated list system consisting of replicas R_1 and R_2 which initially hold the same list (Figure 2). Suppose that user 1 invokes $o_1 = \textsc{Ins}(f, 1)$ at R_1 and concurrently user 2 invokes $o_2 = \textsc{Del}(5)$ at R_2. After being executed locally, each operation is sent to the other replica. Without OT (Figure 2a), the states of two replicas diverge. With the OT of o_1 and o_2 (Figure 2b), o_2 is transformed to $o_2' = \textsc{Del}(6)$ at R_1, taking into account the fact that o_1 has inserted an element at position 1. Meanwhile, o_1 remains unchanged. As a result, two replicas converge to the same list. We note that although the idea of OT is straightforward, many OT-based protocols for replicated list are hard to understand and some of them have even been shown incorrect with respect to convergence [8, 20, 22].

The major obstacle to proving that Jupiter satisfies the weak list specification is the *mismatch* between the *global property* on all states prescribed by such a specification and the *local view* each replica maintains in the protocol. On the one hand, the weak list specification requires that states across the system are pairwise compatible [5]. That is, for any pair of (list) states, there cannot be two elements a and b such that a precedes b in one state but b precedes a in the other. On the other hand, Jupiter uses data structures like 1D buffer [18] or 2D state space [13, 26] which are not "compact" enough to capture all replica states in one. In particular, Jupiter maintains $2n$ 2D state spaces for a system with n clients [26]: Each client maintains a single state space which is synchronized with those of other clients via its counterpart state space maintained by the server. Each 2D state space of a client (as well as its counterpart at the server) consists of a local dimension and a global dimension, keeping track of the operations processed by the client itself and the others, respectively. In this way, replica states of Jupiter are dispersed in multiple 2D state spaces maintained locally at individual replicas.

To resolve the mismatch, we propose CJupiter (Compact Jupiter), a variant of Jupiter, which uses a novel data structure called *n-ary ordered state space* for a system with n clients. CJupiter is compact in the sense that at a high level, it maintains only a single n-ary ordered state space which encompasses exactly all states of each replica. Each replica behavior

[2] Since replicas are required to respond to user operations immediately, the client/server architecture does not imply that clients process operations in the same order.

[3] The details about Figure 1 will be described in Examples 4 and 13.

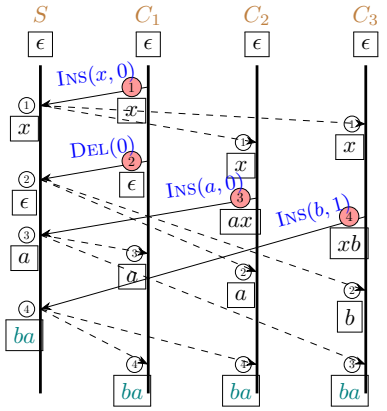
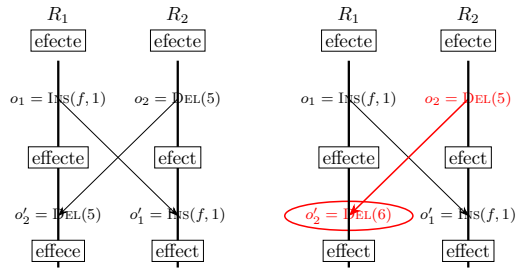


Figure 1 A schedule of four operations adapted from [5], involving a server replica s and three client replicas c_1, c_2, and c_3. The circled numbers indicate the order in which the operations are received at the server. The list contents produced by CJupiter (Section 3) are shown in boxes.



(a) Without OT, the states of R_1 and R_2 diverge.

(b) With OT, R_1 and R_2 converge to the same state.

Figure 2 Illustrations of OT (adapted from [9]).

corresponds to a path going through this state space. This makes it feasible for us to reason about global properties and finally prove that Jupiter satisfies the weak list specification, thus solving the conjecture of Attiya et al. The roadmap is as follows:

- (Section 3) We propose CJupiter based on the n-ary ordered state space data structure.
- (Section 4) We prove that CJupiter is equivalent to Jupiter in the sense that the behaviors of corresponding replicas of these two protocols are the same under the same schedule of operations. Jupiter is slightly optimized in implementation at clients (but not at the server) by eliminating redundant OTs, which, however, has obscured the similarities among clients and led to the mismatch discussed above.
- (Section 5) We prove that CJupiter satisfies the weak list specification. Thanks to the "compactness" of CJupiter, we are able to focus on a single n-ary ordered state space which provides a global view of all possible replica states.

Section 2 presents preliminaries on specifying replicated list data type and OT. Section 6 describes related work. Section 7 concludes the paper. The full paper [25] contains proofs and pseudocode.

2 Preliminaries: Replicated List and Operational Transformation

We describe the system model and specifications of replicated list in the framework for specifying replicated data types [7, 6, 5].

2.1 System Model

A highly-available replicated data store consists of *replicas* that process user operations on the replicated objects and communicate updates to each other with messages. To be *highly-available*, replicas are required to respond to user operations immediately without any communication with others. A *replica* is defined as a state machine $R = (\Sigma, \sigma_0, E, \Delta)$, where *1)* Σ is a set of states; *2)* $\sigma_0 \in \Sigma$ is the initial state; *3)* E is a set of possible events; and *4)* $\Delta : \Sigma \times E \to \Sigma$ is a transition function. The state transitions determined by Δ are local

steps of a replica, describing how it interacts with the following three kinds of *events* from users and other replicas:

- do(o, v): a user invokes an operation $o \in \mathcal{O}$ on the replicated object and immediately receives a response $v \in \mathit{Val}$. We leave the users unspecified and say that the replica *generates* the operation o;
- send(m): the replica sends a message m to some replicas; and
- receive(m): the replica receives a message m.

A *protocol* is a collection $\mathcal{R}$ of replicas. An *execution* α of a protocol $\mathcal{R}$ is a sequence of all events occurring at the replicas in $\mathcal{R}$. We denote by $R(e)$ the replica at which an event e occurs. For an execution (or generally, an event sequence) α, we denote by $e \prec_\alpha e'$ (or $e \prec e'$) that e precedes e' in α. An execution α is *well-formed* if for every replica R: *1)* the subsequence of events $\langle e_1, e_2, \ldots \rangle$ at R, denoted $\alpha|_R$, is well-formed, namely there is a sequence of states $\langle \sigma_1, \sigma_2, \ldots \rangle$, such that $\sigma_i = \Delta(\sigma_{i-1}, e_i)$ for all i; and *2)* every receive(m) event at R is preceded by a send(m) event in α. We consider only *well-formed* executions.

We are often concerned with replica behaviors and states when studying a protocol. The *behavior* of replica R in α is a sequence of the form: $\sigma_0, e_1, \sigma_1, e_2, \ldots$, where $\langle e_1, e_2, \ldots \rangle = \alpha|_R$ and $\sigma_i = \Delta(\sigma_{i-1}, e_i)$ for all i. A *replica state* σ of R in α can be represented by the events in a prefix of $\alpha|_R$ it has processed. Specifically, $\sigma_0 = \langle\rangle$ and $\sigma_i = \sigma_{i-1} \circ e_i = \langle e_1, e_2, \ldots, e_i \rangle$.

We now define the causally-before, concurrent, and totally-before relations on events in an execution. When restricted to the do events only, they define relations on user operations. In an execution α, event e is *causally before* e', denoted $e \xrightarrow{\mathrm{hb}_\alpha} e'$ (or $e \xrightarrow{\mathrm{hb}} e'$), if one of the following conditions holds [10]: *1)* Thread of execution: $R(e) = R(e') \wedge e \prec_\alpha e'$; *2)* Message delivery: $e = \mathrm{send}(m) \wedge e' = \mathrm{receive}(m)$; *3)* Transitivity: $\exists e'' \in \alpha : e \xrightarrow{\mathrm{hb}_\alpha} e'' \wedge e'' \xrightarrow{\mathrm{hb}_\alpha} e'$. Events $e, e' \in \alpha$ are *concurrent*, denoted $e \parallel_\alpha e'$ (or $e \parallel e'$), if it is neither $e \xrightarrow{\mathrm{hb}_\alpha} e'$ nor $e' \xrightarrow{\mathrm{hb}_\alpha} e$. A relation on events in an execution α, denoted $e \xrightarrow{\mathrm{tb}_\alpha} e'$ (or $e \xrightarrow{\mathrm{tb}} e'$), is a *totally-before* relation *consistent with* the causally-before relation '$\xrightarrow{\mathrm{hb}_\alpha}$' on events in α if it is total: $\forall e, e' \in \alpha : e \xrightarrow{\mathrm{tb}_\alpha} e' \vee e' \xrightarrow{\mathrm{tb}_\alpha} e$, and it is consistent: $\forall e, e' \in \alpha : e \xrightarrow{\mathrm{hb}_\alpha} e' \implies e \xrightarrow{\mathrm{tb}_\alpha} e'$.

2.2 Specifying Replicated Objects

A replicated object is specified by a set of abstract executions which record user operations (corresponding to do events) and visibility relations on them [7]. An *abstract execution* is a pair $A = (H, \mathrm{vis})$, where H is a sequence of do events and $\mathrm{vis} \subseteq H \times H$ is an acyclic *visibility* relation such that *1)* if $e_1 \prec_H e_2$ and $R(e_1) = R(e_2)$, then $e_1 \xrightarrow{\mathrm{vis}} e_2$; *2)* if $e_1 \xrightarrow{\mathrm{vis}} e_2$, then $e_1 \prec_H e_2$; and *3)* vis is transitive: $(e_1 \xrightarrow{\mathrm{vis}} e_2 \wedge e_2 \xrightarrow{\mathrm{vis}} e_3) \implies e_1 \xrightarrow{\mathrm{vis}} e_3$.

An abstract execution $A' = (H', \mathrm{vis}')$ is a *prefix* of another abstract execution $A = (H, \mathrm{vis})$ if H' is a prefix of H and $\mathrm{vis}' = \mathrm{vis} \cap (H' \times H')$. A *specification* $\mathcal{S}$ of a replicated object is a *prefix-closed* set of abstract executions, namely if $A \in \mathcal{S}$, then $A' \in \mathcal{S}$ for each prefix A' of A. A protocol $\mathcal{R}$ *satisfies* a specification $\mathcal{S}$, denoted $\mathcal{R} \models \mathcal{S}$, if any (concrete) execution α of $\mathcal{R}$ *complies with* some abstract execution $A = (H, \mathrm{vis})$ in $\mathcal{S}$, namely $\forall R \in \mathcal{R} : H|_R = \alpha|_R^{\mathrm{do}}$, where $\alpha|_R^{\mathrm{do}}$ is the subsequence of do events of replica R in α.

2.3 Replicated List Specification

A replicated list object supports three types of user operations [5] (U for some universe):

- Ins(a, p): inserts $a \in U$ at position $p \in \mathbb{N}$ and returns the updated list. For p larger than the list size, we assume an insertion at the end. We assume that all inserted elements are unique, which can be achieved by attaching replica identifiers and sequence numbers.

- $\textsc{Del}(a, p)$: deletes an element at position $p \in \mathbb{N}$ and returns the updated list. For p larger than the list size, we assume an deletion at the end. The parameter $a \in U$ is used to record the deleted element [22], which will be referred to in condition 1(a) of the weak list specification defined later.
- $\textsc{Read}$: returns the contents of the list.

The operations above, as well as a special NOP (i.e., "do nothing"), form $\mathcal{O}$ and all possible list contents form *Val*. $\textsc{Ins}$ and $\textsc{Del}$ are collectively called *list updates*. We denote by $\text{elems}(A) = \{a \mid \text{do}(\textsc{Ins}(a, \_), \_) \in H\}$ the set of all elements inserted into the list in an abstract execution $A = (H, \text{vis})$.

We adopt the convergence property in [5] which requires that two $\textsc{Read}$ operations that observe the same set of list updates return the same response. Formally, an abstract execution $A = (H, \text{vis})$ belongs to the *convergence property* $\mathcal{A}_{\text{cp}}$ if and only if for any pair of $\textsc{Read}$ events $e_1 = \text{do}(\textsc{Read}, w_1 \triangleq a_1^0 \dots a_1^{m-1})$ and $e_2 = \text{do}(\textsc{Read}, w_2 \triangleq a_2^0 \dots a_2^{n-1})$ $(a_i^j \in \text{elems}(A))$, it holds that $\left(\text{vis}^{-1}_{\textsc{Ins},\textsc{Del}}(e_1) = \text{vis}^{-1}_{\textsc{Ins},\textsc{Del}}(e_2)\right) \implies w_1 = w_2$, where $\text{vis}^{-1}_{\textsc{Ins},\textsc{Del}}(e)$ denotes the set of list updates visible to e.

The weak list specification requires the ordering between elements that are not deleted to be consistent across the system [5].

▶ **Definition 1** (Weak List Specification $\mathcal{A}_{\text{weak}}$ [5])**.** An abstract execution $A = (H, \text{vis})$ belongs to the *weak list specification* $\mathcal{A}_{\text{weak}}$ if and only if there is a relation $\text{lo} \subseteq \text{elems}(A) \times \text{elems}(A)$, called the *list order*, such that:

1. Each event $e = \text{do}(o, w) \in H$ returns a sequence of elements $w = a_0 \dots a_{n-1}$, where $a_i \in \text{elems}(A)$, such that:
   - **a.** w contains exactly the elements visible to e that have been inserted, but not deleted:
   $$\forall a.\, a \in w \iff \left(\text{do}(\textsc{Ins}(a, \_), \_) \le_{\text{vis}} e\right) \wedge \neg\left(\text{do}(\textsc{Del}(a, \_), \_) \le_{\text{vis}} e\right).$$
   - **b.** The list order is consistent with the order of the elements in w:
   $$\forall i, j.\, (i < j) \implies (a_i, a_j) \in \text{lo}.$$
   - **c.** Elements are inserted at the specified position: $op = \textsc{Ins}(a, k) \implies a = a_{\min\{k, n-1\}}$.
2. lo is irreflexive and for all events $e = \text{do}(op, w) \in H$, it is transitive and total on $\{a \mid a \in w\}$.

▶ **Example 2** (Weak List Specification)**.** In the execution depicted in Figure 1 (produced by CJupiter), there exist three states with list contents $w_1 = ba$, $w_2 = ax$, and $w_3 = xb$, respectively. This is allowed by the weak list specification with the list order lo: $b \xrightarrow{\text{lo}} a$ on w_1, $a \xrightarrow{\text{lo}} x$ on w_2, and $x \xrightarrow{\text{lo}} b$ on w_3. However, an execution is not allowed by the weak list specification if it contained two states with, say $w = ab$ and $w' = ba$.

2.4 Operational Transformation (OT)

The OT of transforming $o_1 \in \mathcal{O}$ with $o_2 \in \mathcal{O}$ is expressed by the function $o_1' = OT(o_1, o_2)$. We also write $(o_1', o_2') = OT(o_1, o_2)$ to denote both $o_1' = OT(o_1, o_2)$ and $o_2' = OT(o_2, o_1)$. To ensure the convergence property, OT functions are required to satisfy CP1 (Convergence Property 1) [8]: Given two operations o_1 and o_2, if $(o_1', o_2') = OT(o_1, o_2)$, then $\sigma; o_1; o_2' = \sigma; o_2; o_1'$ should hold, meaning that the same state is obtained by applying o_1 and o_2' in sequence, and applying o_2 and o_1' in sequence, on the same initial state σ. A set of OT functions satisfying CP1 for a replicated list object [8, 9, 22] can be found in Figure A.1 of [25].

3 The CJupiter Protocol

In this section we propose CJupiter (Compact Jupiter) for a replicated list based on the data structure called n-ary ordered state space. Like Jupiter, CJupiter also adopts a client/server architecture. For convenience, we assume that the server does not generate operations [26, 5]. It mainly serializes operations and propagates them from one client to others. We denote by '$\prec_s$' the total order on the set of operations established by the server. Note that '$\prec_s$' is consistent with the causally-before relation '$\xrightarrow{\text{hb}}$'. To facilitate the comparison of Jupiter and CJupiter, we refer to '$\xrightarrow{\text{hb}}$' and '$\prec_s$' together as the *schedule* of operations.

3.1 Data Structure: n-ary Ordered State Space

For a client/server system with n clients, CJupiter maintains $(n+1)$ n-ary ordered state spaces, one per replica (CSS_s for the server and CSS_{c_i} for client c_i). Each CSS is a directed graph whose vertices represent states and edges are labeled with operations; see Appendix B.1 of [25].

An ***operation*** op of type Op is a tuple $op = (o, oid, ctx, sctx)$, where *1)* o is the signature of type $\mathcal{O}$ described in Section 2.3; *2)* oid is a globally unique operation identifier which is a pair (cid, seq) consisting of the client id and a sequence number; *3)* ctx is an *operation context* which is a set of oids, denoting the operations that are causally before op; and *4)* $sctx$ is a set of oids, denoting the operations that, as far as op knows, have been executed before op at the server. At a given replica, $sctx$ is used to determine the total order '$\prec_s$' relation between two operations as in Algorithm B.1 of [25].

The OT function of two operations $op, op' \in Op$, denoted $(op\langle op'\rangle : Op, op'\langle op\rangle : Op) = OT(op, op')$, is defined based on that of $op.o, op'.o \in \mathcal{O}$, denoted $(o, o') = OT(op.o, op'.o)$, such that $op\langle op'\rangle = (o, op.oid, op.ctx \cup \{op'.oid\}, op.sctx)$ and $op'\langle op\rangle = (o', op'.oid, op'.ctx \cup \{op.oid\}, op'.sctx)$.

A ***vertex*** v of type $Vertex$ is a pair $v = (oids, edges)$, where $oids$ is the set of operations (represented by their identifies) that have been executed, and $edges$ is an *ordered* set of edges of type $Edge$ from v to other vertices, labeled with operations. That is, each ***edge*** is a pair $(op : Op, v : Vertex)$. Edges from the same vertex are *totally ordered* by their op components. For each vertex v and each edge $e = (op, u)$ from v to u, it is required that

- the ctx of op associated with e matches the $oids$ of v: $op.ctx = v.oids$;
- the $oids$ of u consists of the $oids$ of v and the oid of op: $u.oids = v.oids \cup \{op.oid\}$.

▶ **Definition 3** (n-ary Ordered State Space)**.** An ***n-ary ordered state space*** is a set of vertices such that

1. Vertices are uniquely identified by their $oids$.
2. For each vertex u with $|u.edges| \geq 2$, let u' be its child vertex along the ***first*** edge $e_{uu'} = (op', u')$ and v another child vertex along $e_{uv} = (op, v)$. There exist (Figure 3)
   - a vertex v' with $v'.oids = u.oids \cup \{op'.oid, op.oid\}$;
   - two edges $e_{u'v'} = (op\langle op'\rangle, v')$ from u' to v' and $e_{vv'} = (op'\langle op\rangle, v')$ from v to v'.

The second condition models OTs in CJupiter described in Section 3.2, and the choice of the "first" edge is justified in Lemmas 5 and 7.

3.2 The CJupiter Protocol

Each replica in CJupiter maintains an n-ary ordered state space S and keeps the most recent vertex cur (initially $(\emptyset, \emptyset)$) of S. Following [26], we describe CJupiter in three parts; see Appendix B.2 of [25] for pseudocode.

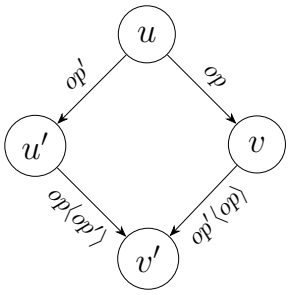


Figure 3 Illustration of an OT of two operations op, op' in both the n-ary ordered state space of CJupiter and the 2D state space of Jupiter: $(op\langle op'\rangle, op'\langle op\rangle) = OT(op, op')$. In the CJupiter and Jupiter protocols (and Examples 4 and 13), op corresponds to the new incoming operation to be transformed.

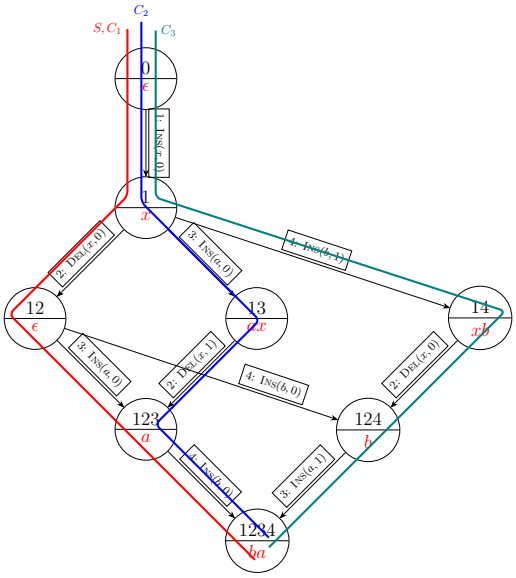


Figure 4 The same final n-ary ordered state space (thus for CSS_s and each CSS_{c_i}) constructed by CJupiter for each replica under the schedule of Figure 1. Each replica behavior (i.e., the sequence of state transitions) corresponds to a path going through this state space.

Local Processing Part. When a client receives an operation $o \in \mathcal{O}$ from a user, it

1. applies o locally, obtaining a new list $val \in Val$;
2. generates $op \in Op$ by attaching to o a unique operation identifier and the operation context $S.cur.oids$, representing the set of operations that are causally before op;
3. creates a vertex v with $v.oids = S.cur.oids \cup \{op.oid\}$, appends v to S by linking it to $S.cur$ via an edge labeled with op, and updates cur to be v;
4. sends op to the server asynchronously and returns val to the user.

Server Processing Part. To establish the total order '$\prec_s$' on operations, the server maintains the set $soids$ of operations it has executed. When the server receives an operation $op \in Op$ from client c_i, it

1. updates $op.sctx$ to be $soids$ and updates $soids$ to include $op.oid$;
2. transforms op with an operation sequence in S to obtain op' by calling $S.\text{xForm}(op)$ (see below), and applies op' (specifically, $op'.o$) locally;
3. sends op (instead of op') to other clients asynchronously.

Remote Processing Part. When a client receives an operation $op \in Op$ from the server, it transforms op with an operation sequence in S to obtain op' by calling $S.\text{xForm}(op)$ (see below), and applies op' (specifically, $op'.o$) locally.

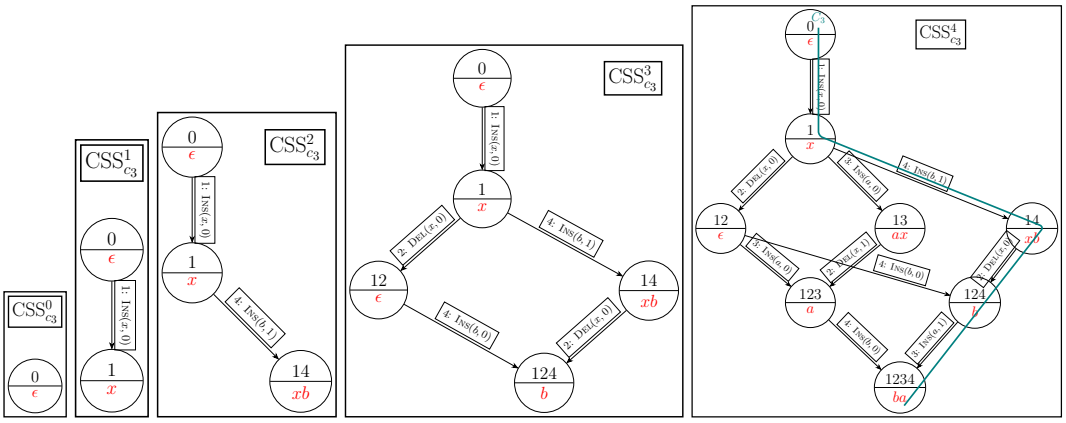


Figure 5 Illustration of client c_3 in CJupiter under the schedule of Figure 1. Its behavior (i.e., the sequence of state transitions) is indicated by the path in $\mathrm{CSS}^4_{c_3}$. (Please refer to Figure B.1 of [25] for the illustration of clients c_1 and c_2 and the server s.)

OTs in CJupiter. The procedure $S.\textsc{xForm}(op : Op)$ transforms op with an operation sequence in an n-ary ordered state space S. Specifically, it

1. locates the vertex u whose $oids$ matches the ctx of op, i.e., $u.oids = op.ctx$ [4], and creates a vertex v with $v.oids = u.oids \cup \{op.oid\}$;
2. iteratively transforms op with an operation sequence consisting of operations along the ***first*** edges from u to the final vertex cur of S (Figure 3):
   a. obtains the vertex u' and the operation op' associated with the first edge of u;
   b. transforms op with op' to obtain $op\langle op'\rangle$ and $op'\langle op\rangle$;
   c. creates a vertex v' with $v'.oids = v.oids \cup \{op'.oid\}$;
   d. links v' to v via an edge labeled with $op'\langle op\rangle$ and v to u via an edge labeled with op;
   e. updates u, v, and op to be u', v', and $op\langle op'\rangle$, respectively;
3. when u is the final vertex cur of S, links v to u via an edge labeled with op, updates cur to be v, and returns the last transformed operation op.

To keep track of the construction of the n-ary ordered state spaces in CJupiter, for each state space, we introduce a superscript k to refer to the one after the k-th step (i.e., after processing k operations), counting from 0. For instance, the state space CSS_{c_i} (resp. CSS_s) after the k-th step maintained by client c_i (resp. the server s) is denoted by $\mathrm{CSS}^k_{c_i}$ (resp. CSS^k_s). This notational convention also applies to Jupiter (reviewed in Section 4.1).

▶ **Example 4** (Illustration of CJupiter). Figure 5 illustrates client c_3 in CJupiter under the schedule of Figure 1. For convenience, we denote, for instance, a vertex v with $v.oids = \{o_1, o_4\}$ by v_{14} and an operation o_3 with $o_3.ctx = \{o_1, o_2\}$ by $o_3\{o_1, o_2\}$. We have also mixed the notations of operations of types $\mathcal{O}$ and Op when no confusion arises. We map various vertices and operations in this example to the ones (i.e., u, u', v, v', op, op') used in the description of the CJupiter protocol.

After receiving and applying $o_1 = \textsc{Ins}(x, 0)$ of client c_1 from the server, client c_3 generates $o_4 = \textsc{Ins}(b, 1)$. It applies o_4 locally, creates a new vertex v_{14}, and appends it to $\mathrm{CSS}^1_{c_3}$ via an edge from v_1 labeled with $o_4\{o_1\}$. Then, $o_4\{o_1\}$ is propagated to the server.

[4] The vertex u exists due to the FIFO communication between the clients and the server.

Next, client c_3 receives $o_2 = \textsc{Del}(x, 0)$ of client c_1 from the server. The operation context of o_2 is $\{o_1\}$, matching the *oids* of v_1 (u). By xForm, $o_2\{o_1\}$ (op) is transformed with $o_4\{o_1\}$ (op'): $OT\Big(o_2\{o_1\} = \textsc{Del}(x, 0), o_4\{o_1\} = \textsc{Ins}(b, 1)\Big) = \Big(o_2\{o_1, o_4\} = \textsc{Del}(x, 0), o_4\{o_1, o_2\} = \textsc{Ins}(b, 0)\Big)$. As a result, v_{124} (v') is created and is linked to v_{12} (v) and v_{14} (u') via the edges labeled with $o_4\{o_1, o_2\}$ and $o_2\{o_1, o_4\}$, respectively. Because o_2 is unaware of o_4 at the server ($o_4.sctx = \emptyset$ now), the edge from v_1 to v_{12} is ordered before (to the left of) that from v_1 to v_{14} in $\text{CSS}^3_{c_3}$.

Finally, client c_3 receives $o_3\{o_1\} = \textsc{Ins}(a, 0)$ of client c_2 from the server. The operation context of o_3 is $\{o_1\}$, matching the *oids* of v_1 (u). By xForm, $o_3\{o_1\}$ will be transformed with the operation sequence consisting of operations along the *first* edges from v_1 to the final vertex v_{124} of $\text{CSS}^3_{c_3}$, namely $o_2\{o_1\}$ from v_1 and $o_4\{o_1, o_2\}$ from v_{12}. Specifically, $o_3\{o_1\}$ (op) is first transformed with $o_2\{o_1\}$ (op'): $OT\Big(o_3\{o_1\} = \textsc{Ins}(a, 0), o_2\{o_1\} = \textsc{Del}(x, 0)\Big) = \Big(o_3\{o_1, o_2\} = \textsc{Ins}(a, 0), o_2\{o_1, o_3\} = \textsc{Del}(x, 1)\Big)$. Since o_3 is aware of o_2 but unaware of o_4 at the server, the new edge from v_1 labeled with $o_3\{o_1\}$ is placed before that with $o_4\{o_1\}$ but after that with $o_2\{o_1\}$. Then, $o_3\{o_1, o_2\}$ (op) is transformed with $o_4\{o_1, o_2\}$ (op'), yielding v_{1234} and $o_3\{o_1, o_2, o_4\}$. Client c_3 applies $o_3\{o_1, o_2, o_4\}$, obtaining the list content ba.

The choice of the "first" edges in OTs is necessary to establish equivalence between CJupiter and Jupiter, particularly *at the server side*. First, the operation sequence along the first edges from a vertex of CSS_s at the server admits a simple characterization.

▶ **Lemma 5** (CJupiter's "First" Rule). *Let $OP = \langle op_1, op_2, \ldots, op_m\rangle$ ($op_i \in Op$) be the operation sequence the server has currently processed in total order '$\prec_s$'. For any vertex v in the current CSS_s, the path along the **first** edges from v to the final vertex of CSS_s consists of the operations of $OP \setminus v$ in total order '$\prec_s$' (may be empty if v is the final vertex of CSS_s), where*

$$OP \setminus v = \Big\{op \in OP \mid op.oid \in \{op_1.oid, op_2.oid, \cdots, op_m.oid\} \setminus v.oids\Big\}.$$

▶ **Example 6** (CJupiter's "First" Rule). Consider CSS_s at the server shown in Figure 4 under the schedule of Figure 1; see Figure B.1a of [25] for its construction. Suppose that the server has processed all four operations. That is, we take $OP = \langle o_1, o_2, o_3, o_4\rangle$ in Lemma 5 (we mix operations of types $\mathcal{O}$ and Op). Then, the path along the first edges from vertex v_1 (resp. v_{13}) consists of the operations $OP \setminus v_1 = \{o_2, o_3, o_4\}$ (resp. $OP \setminus v_{13} = \{o_2, o_4\}$) in total order '$\prec_s$'.

Based on Lemma 5, the operation sequence with which an operation transforms *at the server* can be characterized as follows, which is exactly the same with that for Jupiter [26].

▶ **Lemma 7** (CJupiter's OT Sequence). *In* xForm *of* CJupiter*, the operation sequence L (may be empty) with which an operation op transforms **at the server** consists of the operations that are both totally ordered by '$\prec_s$' before and concurrent by '$\|$' with op. Furthermore, the operations in L are totally ordered by '$\prec_s$'.*

▶ **Example 8** (CJupiter's OT Sequence). Consider the behavior of the server summarized in Figure 4 under the schedule of Figure 1. According to Lemma 5, the operation sequence with which $op = o_4$ transforms consists of operations o_2 (i.e., $o_2\{o_1\}$) from vertex v_1 and o_3 (i.e., $o_3\{o_1, o_2\}$) from vertex v_{12} in total order '$\prec_s$', which are both totally ordered by '$\prec_s$' before and concurrent by '$\|$' with o_4.

3.3 CJupiter is Compact

Although $(n+1)$ n-ary ordered state spaces are maintained by CJupiter for a system with n clients, they are all the same. That is, at a high level, CJupiter maintains only a single n-ary ordered state space.

▶ **Proposition 9** ($n+1 \Rightarrow 1$)**.** *In* CJupiter*, the replicas that have processed the same set of operations (in terms of their oids) have the same n-ary ordered state space.*

Informally, this proposition holds because we have kept all "by-product" states/vertices of OTs in the n-ary ordered state spaces, and each client is "synchronized" with the server. Since all replicas will eventually process all operations, the final n-ary ordered state spaces at all replicas are the same. The construction order may differ replica by replica.

▶ **Example 10** (CJupiter is Compact)**.** Figure 4 shows the same final n-ary ordered state space constructed by CJupiter for each replica under the schedule of Figure 1. (Figure B.1 of [25] shows the step-by-step construction for each replica.) Each replica behavior (i.e., the sequence of state transitions) corresponds to a path going through this state space. As illustrated, the server s and client c_1 go along the path $v_0 \xrightarrow{o_1} v_1 \xrightarrow{o_2} v_{12} \xrightarrow{o_3} v_{123} \xrightarrow{o_4} v_{1234}$, client c_2 goes along the path $v_0 \xrightarrow{o_1} v_1 \xrightarrow{o_3} v_{13} \xrightarrow{o_2} v_{123} \xrightarrow{o_4} v_{1234}$, and client c_3 goes along the path $v_0 \xrightarrow{o_1} v_1 \xrightarrow{o_4} v_{14} \xrightarrow{o_2} v_{124} \xrightarrow{o_3} v_{1234}$.

Together with the fact that the OT functions satisfy CP1, Proposition 9 implies that

▶ **Theorem 11** (CJupiter $\models \mathcal{A}_{\mathsf{cp}}$)**.** CJupiter *satisfies the convergence property $\mathcal{A}_{cp}$.*

4 CJupiter is Equivalent to Jupiter

We now prove that CJupiter is equivalent to Jupiter (reviewed in Section 4.1) from perspectives of both the server and clients. Specifically, we prove that the behaviors of the servers are the same (Section 4.2), and that the behaviors of each pair of corresponding clients are the same (Section 4.3). Consequently, we have that

▶ **Theorem 12** (Equivalence)**.** *Under the same schedule, the behaviors (Section 2.1) of corresponding replicas in* CJupiter *and* Jupiter *are the same.*

4.1 Review of Jupiter

We review the Jupiter protocol in [26], a *multi-client* description of Jupiter first proposed in [13] [5]. Consider a client/server system with n clients. Jupiter [26] maintains $2n$ *2D state spaces* (Appendix C.1 of [25]), each consisting of a *local* dimension and a *global* dimension. Specifically, each client c_i maintains a 2D state space, denoted DSS_{c_i}, with the local dimension for operations generated by the client and the global dimension by others. The server maintains n 2D state spaces, one for each client. The state space for client c_i, denoted DSS_{s_i}, consists of the local dimension for operations from client c_i and the global dimension from others.

Jupiter is similar to CJupiter with two major differences: First, in $\textsc{xform}(op : Op, d \in \{LOCAL, GLOBAL\})$ of Jupiter, the operation sequence with which op transforms is determined by the parameter d, indicating the local/global dimension described above (instead of following

[5] The Jupiter protocol in [13] uses 1D buffers, but does not explicitly describe the multi-client scenario.

the *first* edges as in CJupiter). Second, in Jupiter, the server propagates the *transformed* operation (instead of the original one it receives) to other clients. As with CJupiter, we describe Jupiter in three parts. We omit the details that are in common with and have been explained in CJupiter; see Appendix C.2 of [25] for pseudocode.

Local Processing Part. When client c_i receives an operation $o \in \mathcal{O}$ from a user, it applies o locally, generates $op \in Op$ for o, saves op along the local dimension at the end of its 2D state space DSS_{c_i}, and sends op to the server asynchronously.

Server Processing Part. When the server receives an operation $op \in Op$ from client c_i, it first transforms op with an operation sequence along the global dimension in DSS_{s_i} to obtain op' by calling $\textsc{xForm}(op, \mathit{GLOBAL})$ (see below), and applies op' locally. Then, for each $j \neq i$, it saves op' at the end of DSS_{s_j} along the global dimension. Finally, op' (instead of op) is sent to other clients asynchronously.

Remote Processing Part. When client c_i receives an operation $op \in Op$ from the server, it transforms op with an operation sequence along the local dimension in its 2D state space DSS_{c_i} to obtain op' by calling $\textsc{xForm}(op, \mathit{LOCAL})$ (see below), and applies op' locally.

OTs in Jupiter. In the procedure $\textsc{xForm}(op : Op, d : LG = \{\mathit{LOCAL}, \mathit{GLOBAL}\})$ of Jupiter, the operation sequence with which op transforms is determined by an extra parameter d. Specifically, it first locates the vertex u whose *oids* matches the operation context $op.ctx$ of op, and then iteratively transforms op with an operation sequence along the d dimension from u to the final vertex of this 2D state space.

▶ **Example 13** (Illustration of Jupiter). Figure 6 illustrates client c_3, as well as the server s, in Jupiter under the schedule of Figure 1. The first three state transitions made by client c_3 in Jupiter due to the operation sequence consisting of o_1 from client c_1, o_4 generated by itself, and o_2 from client c_1 are the same with those in CJupiter; see $\mathrm{CSS}^1_{c_3}$, $\mathrm{CSS}^2_{c_3}$, and $\mathrm{CSS}^3_{c_3}$ of Figure 5 and $\mathrm{DSS}^1_{c_3}$, $\mathrm{DSS}^2_{c_3}$, and $\mathrm{DSS}^3_{c_3}$ of Figure 6.

We now elaborate on the fourth state transition of client c_3 in Jupiter. First, client c_2 propagates its operation $o_3\{o_1\} = \textsc{Ins}(a, 0)$ to the server s. At the server, $o_3\{o_1\}$ is transformed with $o_2\{o_1\} = \textsc{Del}(x, 0)$ in $\mathrm{DSS}^3_{s_2}$, obtaining $o_3\{o_1, o_2\} = \textsc{Ins}(a, 0)$. In addition to being stored in $\mathrm{DSS}^3_{s_1}$ and $\mathrm{DSS}^3_{s_3}$, the transformed operation $o_3\{o_1, o_2\}$ is then redirected by the server to clients c_1 and c_3. At client c_3, the operation context of $o_3\{o_1, o_2\}$ (i.e., $\{o_1, o_2\}$) matches the *oids* of v_{12} (u) in $\mathrm{DSS}^4_{c_3}$. By $\textsc{xForm}$, $o_3\{o_1, o_2\}$ (op) is transformed with $o_4\{o_1, o_2\}$ (op'), yielding v_{1234} and $o_3\{o_1, o_2, o_4\}$. Finally, client c_3 applies $o_3\{o_1, o_2, o_4\}$, obtaining the list content ba.

We highlight three differences between CJupiter and Jupiter, by comparing the behaviors of client c_3 in this example and Example 4. First, the fourth operation the server s redirects to client c_3 is the transformed operation $o_3\{o_1, o_2\} = \textsc{Ins}(a, 0)$, instead of the original one $o_3\{o_1\} = \textsc{Ins}(a, 0)$ [6] generated by client c_2. Second, each vertex in the n-ary ordered state space of CJupiter (such as $\mathrm{CSS}^4_{c_3}$ of Figure 5) is not restricted to have only two child vertices, while Jupiter does. Third, because the transformed operations are propagated by the server, Jupiter is slightly optimized in implementation *at clients* by eliminating redundant OTs. For example, in $\mathrm{CSS}^4_{c_3}$ of Figure 5, the original operation $o_3\{o_1\}$ of client c_2 redirected by the

[6] Although they happen to have the same signature $\textsc{Ins}(a, 0)$, they have different operation contexts.

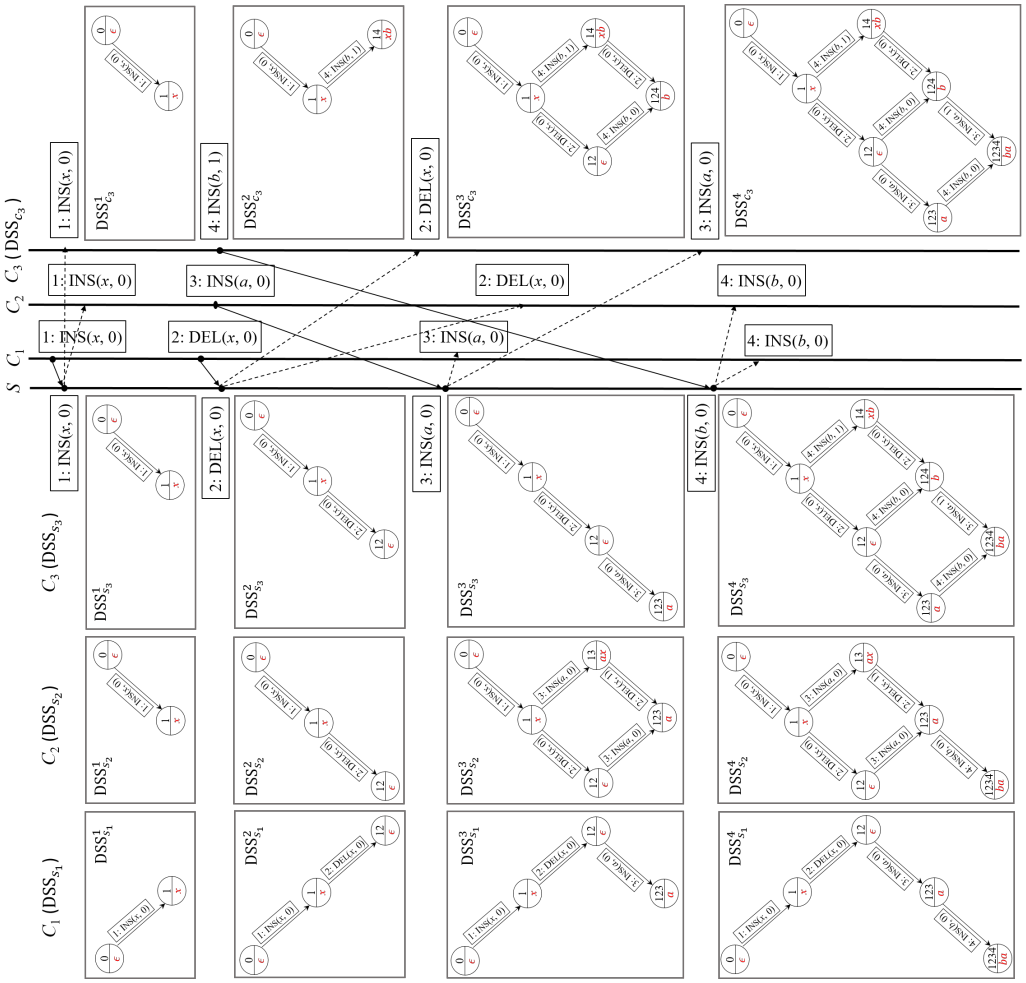


Figure 6 (Rotated) illustration of client c_3, as well as the server s, in Jupiter [26] under the schedule of Figure 1. (Please refer to Figure C.1 of [25] for details of clients c_1 and c_2.)

server should be first transformed with $o_2\{o_1\}$ to obtain $o_3\{o_1, o_2\}$. In Jupiter, however, such a transformation which has been done at the server (i.e., in $\mathrm{DSS}^3_{s_2}$) is not necessary at client c_3 (i.e., in $\mathrm{DSS}^4_{c_3}$).

4.2 The Servers Established Equivalent

As shown in [26] (see the "Jupiter" section and Definition 8 of [26]), the operation sequence with which an incoming operation transforms *at the server* in xForm of Jupiter can be characterized exactly as in xForm of CJupiter (Lemma 7). By mathematical induction on the operation sequence the server processes, we can prove that the state spaces of Jupiter and CJupiter at the server are essentially the same. Formally, the n-ary ordered state space CSS_s of CJupiter equals the union [7] of all 2D state spaces DSS_{s_i} maintained at the server for each client c_i in Jupiter. For example, CSS_s of Figure 4 is the union of the three DSS_{s_i}'s of Figure 6. More specifically, we have

[7] The union is taken on state spaces which are (directed) graphs as sets of vertices and edges. The order of edges of n-ary ordered state spaces should be respected when DSS_{s_i}'s are unioned to obtain CSS_s.

▶ **Proposition 14** ($n \leftrightarrow 1$). *Suppose that under the same schedule, the server has processed a sequence of m operations, denoted $O = \langle op_1, op_2, \ldots, op_m \rangle$ ($op_i \in Op$), in total order '$\prec_s$'. We have that*

$$CSS_s^k = \bigcup_{i=1}^{i=k} DSS^i_{s_{c(op_i)}} = \bigcup_{c_i \in c(O)} \bigcup_{j=1}^{j=k} DSS^j_{s_{c_i}}, \quad 1 \leq k \leq m, \tag{$*$}$$

where $c(op_i)$ denotes the client that generates the operation op_i (more specifically, $op_i.o$) and $c(O) = \{c(op_1), c(op_2), \ldots, c(op_m)\}$.

The equivalence of servers are thus established.

▶ **Theorem 15** (Equivalence of Servers). *Under the same schedule, the behaviors (i.e., the sequence of (list) state transitions, defined in Section 2.1) of the servers in* CJupiter *and* Jupiter *are the same.*

4.3 The Clients Established Equivalent

As discussed in Example 13, Jupiter is slightly optimized in implementation *at clients* by eliminating redundant OTs. Formally, by mathematical induction on the operation sequence client c_i processes, we can prove that $\mathrm{DSS}^k_{c_i}$ of Jupiter is a part (i.e., subgraph) of $\mathrm{CSS}^k_{c_i}$ of CJupiter. The equivalence of clients follows since the final transformed operations (for an original one) executed at c_i in Jupiter and CJupiter are the same, regardless of the optimization adopted by Jupiter at clients.

▶ **Proposition 16** ($1 \leftrightarrow 1$). *Under the same schedule, we have that*

$$DSS^k_{c_i} \subseteq CSS^k_{c_i}, \quad 1 \leq i \leq n,\ k \geq 1. \tag{$\star$}$$

▶ **Theorem 17** (Equivalence of Clients). *Under the same schedule, the behaviors (Section 2.1) of each pair of corresponding clients in* CJupiter *and* Jupiter *are the same.*

5 CJupiter Satisfies the Weak List Specification

The following theorem, together with Theorem 12, solves the conjecture of Attiya et al. [5].

▶ **Theorem 18** (CJupiter $\models \mathcal{A}_{\mathsf{weak}}$). CJupiter *satisfies the weak list specification $\mathcal{A}_{weak}$.*

Proof. For each execution α of CJupiter, we construct an abstract execution $A = (H, \mathrm{vis})$ with $\mathrm{vis} = \xrightarrow{\mathrm{hb}_\alpha}$ (Section 2.1). We then prove the conditions of $\mathcal{A}_{\mathrm{weak}}$ (Definition 1) in the order 1(c), 1(a), 1(b), and 2.

Condition 1(c) follows from the local processing of CJupiter. Condition 1(a) holds due to the FIFO communication and the property of OTs that when transformed in CJupiter, the type and effect of an $\textsc{Ins}(a, p)$ (resp. a $\textsc{Del}(a, p)$) remains unchanged (with a trivial exception of being transformed to be NOP), namely to insert (resp. delete) the element a (possibly at a different position than p).

To show that $A = (H, \mathrm{vis})$ belongs to $\mathcal{A}_{\mathrm{weak}}$, we define the list order relation lo in Definition 19 below, and then prove that lo satisfies conditions 1(b) and 2 of Definition 1. ◀

▶ **Definition 19** (List Order 'lo'). Let α be an execution. For $a, b \in \mathrm{elems}(A)$, $a \xrightarrow{\mathrm{lo}} b$ if and only if there exists an event $e \in \alpha$ with returned list w such that a precedes b in w.

By definition, *1)* lo is *transitive* and *total* on $\{a \mid a \in w\}$ for all events $e = \mathrm{do}(o, w) \in H$; *and 2)* lo satisfies 1(b) of Definition 1. The *irreflexivity* of lo can be rephrased in terms of the pairwise state compatibility property.

▶ **Definition 20** (State Compatibility)**.** Two list states w_1 and w_2 are *compatible*, if and only if for any two common elements a and b of w_1 and w_2, their relative orderings are the same in w_1 and w_2.

▶ **Lemma 21** (Irreflexivity)**.** *Let α be an execution and $A = (H, vis)$ the abstract execution constructed from α as described in the proof of Theorem 18. The list order* lo *based on α is irreflexive if and only if the list states (i.e., returned lists) in A are pairwise compatible.*

The proof relies on the following lemma about paths in n-ary ordered state spaces.

▶ **Lemma 22** (Simple Path)**.** *Let $P_{v_1 \rightsquigarrow v_2}$ be a path from vertex v_1 to vertex v_2 in an n-ary ordered state space. Then, there are no duplicate operations (in terms of their oids) along the path $P_{v_1 \rightsquigarrow v_2}$. We call such a path a simple path.*

Therefore, it remains to prove that all list states in an execution of CJupiter are pairwise compatible, which concludes the proof of Theorem 18. By Proposition 9, we can focus on the state space CSS_s at the server. We first prove several properties about vertex pairs and paths of CSS_s, which serve as building blocks for the proof of the main result (Theorem 26).

By mathematical induction on the operation sequence processed in the total order '$\prec_s$' at the server and by contradiction (in the inductive step), we can show that

▶ **Lemma 23** (LCA)**.** *In* CJupiter*, each pair of vertices in the n-ary ordered state space CSS_s (as a rooted directed acyclic graph) has a unique LCA (Lowest Common Ancestor).* [8]

In the following, we are concerned with the paths to a pair of vertices from their LCA.

▶ **Lemma 24** (Disjoint Paths)**.** *Let v_0 be the unique LCA of a pair of vertices v_1 and v_2 in the n-ary ordered state space CSS_s, denoted $v_0 = LCA(v_1, v_2)$. Then, the set of operations $O_{v_0 \rightsquigarrow v_1}$ along a simple path $P_{v_0 \rightsquigarrow v_1}$ is disjoint in terms of the operation oids from the set of operations $O_{v_0 \rightsquigarrow v_2}$ along a simple path $P_{v_0 \rightsquigarrow v_2}$.*

The next lemma gives a sufficient condition for two states (vertices) being compatible in terms of disjoint simple paths to them from a common vertex.

▶ **Lemma 25** (Compatible Paths)**.** *Let $P_{v_0 \rightsquigarrow v_1}$ and $P_{v_0 \rightsquigarrow v_2}$ be two paths from vertex v_0 to vertices v_1 and v_2, respectively in the n-ary ordered state space CSS_s. If they are disjoint simple paths, then the list states of v_1 and v_2 are compatible.*

The desired pairwise state compatibility property follows, when we take the common vertex v_0 in Lemma 25 as the LCA of the two vertices v_1 and v_2 under consideration.

▶ **Theorem 26** (Pairwise State Compatibility)**.** *Every pair of list states in the state space CSS_s are compatible.*

Proof. Consider vertices v_1 and v_2 in CSS_s. *1)* By Lemma 23, they have a unique LCA, denoted v_0; *2)* By Lemma 22, $P_{v_0 \rightsquigarrow v_1}$ and $P_{v_0 \rightsquigarrow v_2}$ are simple paths; *3)* By Lemma 24, $P_{v_0 \rightsquigarrow v_1}$ and $P_{v_0 \rightsquigarrow v_2}$ are disjoint; and *4)* By Lemma 25, the list states of v_1 and v_2 are compatible. ◀

[8] The LCAs of two vertices v_1 and v_2 in a rooted directed acyclic graph is a set of vertices V such that *1)* Each vertice in V has both v_1 and v_2 as descendants; *2)* In V, no vertice is an ancestor of another. The uniqueness further requires $|V| = 1$.

6 Related Work

Convergence is the main property for implementing a highly-available replicated list object [8, 26]. Since 1989 [8], a number of OT [8]-based protocols have been proposed. These protocols can be classified according to whether they rely on a total order on operations [26]. Various protocols like Jupiter [13, 26] establish a total order via a central server, a sequencer, or a distributed timestamping scheme [1, 24, 18, 12, 23]. By contrast, protocols like adOPTed [15] rely only on a partial (causal) order on operations [8, 14, 21, 20, 19].

In 2016, Attiya et al. [5] propose the strong/weak list specification of a replicated list object. They prove that the existing CRDT (Conflict-free Replicated Data Types) [17]-based RGA protocol [16] satisfies the strong list specification, and *conjecture* that the well-known OT-based Jupiter protocol [13, 26] satisfies the weak list specification.

The OT-based protocols typically use data structures like 1D buffer [18], 2D state space [13, 26], or N-dimensional interaction model [15] to keep track of OTs or choose correct OTs to perform. As a generalization of 2D state space, our n-ary ordered state space is similar to the N-dimensional interaction model. However, they are proposed for different system models. In an n-ary ordered state space, edges from the same vertex are *ordered*, utilizing the existence of a total order on operations. By contrast, the N-dimensional interaction model relies only on a partial order on operations. Consequently, the simple characterization of OTs in xForm of CJupiter does not apply in the N-dimensional interaction model.

7 Conclusion and Future Work

We prove that the Jupiter protocol [13, 26] satisfies the weak list specification [5], thus solving the conjecture recently proposed by Attiya et al. [5]. To this end, we have designed CJupiter based on a novel data structure called n-ary ordered state space. In the future, we will explore how to algebraically manipulate and reason about n-ary ordered state spaces. We also plan to generalize this data structure to the scenarios without a central server and study whether the distributed adOPTed protocol [15] satisfies the strong/weak list specification.

References

1 Apache Wave. `https://incubator.apache.org/wave/`.
2 Google Docs. `https://docs.google.com`.
3 What's different about the new Google Docs: Making collaboration fast. `https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html`.
4 *Apache Wave (incubating) Protocol Documentation (Release 0.4)*, August 22, 2015.
5 Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and complexity of collaborative text editing. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 259–268. ACM, 2016.
6 Hagit Attiya, Faith Ellen, and Adam Morrison. Limitations of Highly-Available Eventually-Consistent Data Stores. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 385–394. ACM, 2015.
7 Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated Data Types: Specification, Verification, Optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284. ACM, 2014.
8 C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 399–407. ACM, 1989.

9 Abdessamad Imine, Michaël Rusinowitch, Gérald Oster, and Pascal Molli. Formal Design and Verification of Operational Transformation Algorithms for Copies Convergence. *Theor. Comput. Sci.*, 351(2):167–183, February 2006.

10 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

11 Bo Leuf and Ward Cunningham. *The Wiki Way: Quick Collaboration on the Web*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

12 Rui Li, Du Li, and Chengzheng Sun. A Time Interval Based Consistency Control Algorithm for Interactive Groupware Applications. In *Proceedings of the 10th International Conference on Parallel and Distributed Systems*, ICPADS '04, pages 429–438, 2004.

13 David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 111–120. ACM, 1995.

14 Atul Prakash and Michael J. Knister. A Framework for Undoing Actions in Collaborative Systems. *ACM Trans. Comput.-Hum. Interact.*, 1(4):295–330, December 1994.

15 Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An Integrating, Transformation-oriented Approach to Concurrency Control and Undo in Group Editors. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, CSCW '96, pages 288–297. ACM, 1996.

16 Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated Abstract Data Types: Building Blocks for Collaborative Applications. *J. Parallel Distrib. Comput.*, 71(3):354–368, March 2011.

17 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400. Springer-Verlag, 2011.

18 Haifeng Shen and Chengzheng Sun. Flexible Notification for Collaborative Systems. In *Proceedings of the 2002 ACM Conference on Computer Supported Cooperative Work*, CSCW '02, pages 77–86. ACM, 2002.

19 Chengzheng Sun. Undo As Concurrent Inverse in Group Editors. *ACM Trans. Comput.-Hum. Interact.*, 9(4):309–361, December 2002.

20 Chengzheng Sun and Clarence Ellis. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pages 59–68. ACM, 1998.

21 Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving Convergence, Causality Preservation, and Intention Preservation in Real-time Cooperative Editing Systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998.

22 Chengzheng Sun, Yi Xu, and Agustina Agustina. Exhaustive Search of Puzzles in Operational Transformation. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work*, CSCW '14, pages 519–529. ACM, 2014.

23 David Sun and Chengzheng Sun. Context-Based Operational Transformation in Distributed Collaborative Editing Systems. *IEEE Trans. Parallel Distrib. Syst.*, 20(10):1454–1470, October 2009.

24 Nicolas Vidot, Michelle Cart, Jean Ferrié, and Maher Suleiman. Copies Convergence in a Distributed Real-time Collaborative Environment. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, CSCW '00, pages 171–180. ACM, 2000.

25 Hengfeng Wei, Yu Huang, and Jian Lu. Specification and Implementation of Replicated List: The Jupiter Protocol Revisited. *CoRR*, abs/1708.04754, 2017.

26 Yi Xu, Chengzheng Sun, and Mo Li. Achieving Convergence in Operational Transformation: Conditions, Mechanisms and Systems. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work*, CSCW '14, pages 505–518. ACM, 2014.