Specification and Implementation of Replicated List: The Jupiter Protocol Revisited

Hengfeng Wei[1]
State Key Laboratory for Novel Software Technology, Nanjing University, China
hfwei@nju.edu.cn
https://orcid.org/0000-0002-0427-9710

Huang Yu
State Key Laboratory for Novel Software Technology, Nanjing University, China
yuhuang@nju.edu.cn
https://orcid.org/0000-0001-8921-036X

Jian Lu
State Key Laboratory for Novel Software Technology, Nanjing University, 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. By orthogonally integrating the n-ary ordered state space with a distributed scheme to totally order operations, we also extend CJupiter to a distributed setting.

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...

Related Version A short version has been accepted as a brief announcement by PODC'2018.

Funding National 973 Program of China (2015CB352202), National Natural Science Foundation of China (61702253)

Acknowledgements We gratefully thank Xue Jiang for drawing Figure 6.

[1] Fundamental Research Funds for the Central Universities (020214380031)

1 Introduction

Collaborative text editing systems, like Google Docs [9], Apache Wave [7], CodoxWord [15], or wikis [13], 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 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 [6, 16, 28, 1]. A common specification of a replicated list object is the *convergence* property, proposed by Ellis et al. [6]. It requires the *final lists at all replicas* be identical after executing the same set of user operations. Recently, Attiya et al. [1] 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. [1] have proved that the existing RGA protocol [19] satisfies the strong list specification. Meanwhile, it is *conjectured* that the well-known Jupiter protocol [16, 28], which is behind Google Docs [10] and Apache Wave [8], satisfies the weak list specification.

Jupiter adopts a *centralized server* replica for propagating updates. [2] Client replicas are connected to the server replica via FIFO channels (Figure 1). Jupiter relies on the technique of operational transformations (OT) [6, 23] 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 [6, 23, 25].

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 [1]. 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 [21] or 2D state space [16, 28] 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 [28]: 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

[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. It turns out to be rather complicated to achieve convergence in spite of a server.

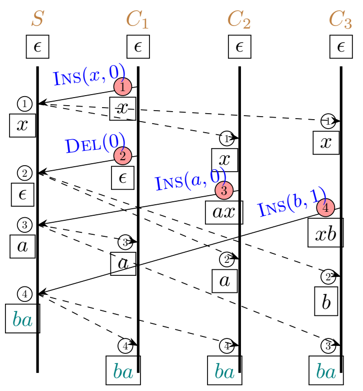
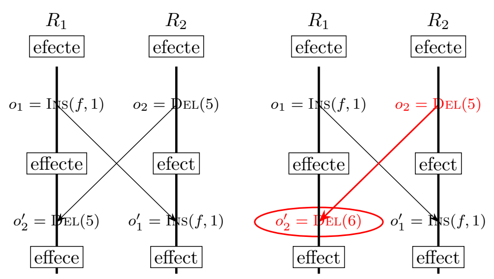


Figure 1 A schedule of four operations adapted from [1], involving a server replica s and three client replicas c_1, c_2, and c_3. The circled numbers indicate the order in which the operations are received at the server. The list contents produced by CJupiter (Section 3) are shown in boxes.



(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 [11]).

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 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
- (Section 6) By integrating the n-ary ordered state space with a distributed scheme (instead of a central server) to totally order operations, we extend CJupiter to a distributed setting. which provides a global view of all possible replica states.

Section 2 presents preliminaries on specifying replicated list data type and OT. Section 7 describes related work. Section 8 concludes the paper. The full protocols and proofs are provided in the appendix.

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 [4, 2, 1].

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:

- $\text{do}(o, v)$: 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;
- $\text{send}(m)$: the replica sends a message m to some replicas; and
- $\text{receive}(m)$: 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 $\text{receive}(m)$ event at R is preceded by a $\text{send}(m)$ 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 $e \xrightarrow{\text{hb}_\alpha} e'$ (or $e \xrightarrow{\text{hb}} e'$), if one of the following conditions holds [12]: *1)* Thread of execution: $R(e) = R(e') \land e \prec_\alpha e'$; *2)* Message delivery: $e = \text{send}(m) \land e' = \text{receive}(m)$; *3)* Transitivity: $\exists e'' \in \alpha : e \xrightarrow{\text{hb}_\alpha} e'' \land e'' \xrightarrow{\text{hb}_\alpha} e'$. Events $e, e' \in \alpha$ are *concurrent*, denoted $e \parallel_\alpha e'$ (or $e \parallel e'$), if it is neither $e \xrightarrow{\text{hb}_\alpha} e'$ nor $e' \xrightarrow{\text{hb}_\alpha} e$. A relation on events in an execution α, denoted $e \xrightarrow{\text{tb}_\alpha} e'$ (or $e \xrightarrow{\text{tb}} e'$), is a *totally-before* relation *consistent with* the causally-before relation '$\xrightarrow{\text{hb}_\alpha}$' on events in α if it is total: $\forall e, e' \in \alpha : e \xrightarrow{\text{tb}_\alpha} e' \lor e' \xrightarrow{\text{tb}_\alpha} e$, and it is consistent: $\forall e, e' \in \alpha : e \xrightarrow{\text{hb}_\alpha} e' \implies e \xrightarrow{\text{tb}_\alpha} e'$.

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 [4]. An *abstract execution* is a pair $A = (H, \text{vis})$, where H is a sequence of do events and $\text{vis} \subseteq H \times H$ is an acyclic *visibility* relation such that *1)* if $e_1 \prec_H e_2$ and $R(e_1) = R(e_2)$, then $e_1 \xrightarrow{\text{vis}} e_2$; *2)* if $e_1 \xrightarrow{\text{vis}} e_2$, then $e_1 \prec_H e_2$; and *3)* vis is transitive: $(e_1 \xrightarrow{\text{vis}} e_2 \land e_2 \xrightarrow{\text{vis}} e_3) \implies e_1 \xrightarrow{\text{vis}} e_3$.

An abstract execution $A' = (H', \text{vis}')$ is a *prefix* of another abstract execution $A = (H, \text{vis})$ if H' is a prefix of H and $\text{vis}' = \text{vis} \cap (H' \times H')$. 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 $A = (H, \text{vis})$ in $\mathcal{S}$, namely $\forall R \in \mathcal{R} : H|_R = \alpha|_R^{\text{do}}$, where $\alpha|_R^{\text{do}}$ 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 [1] (U for some universe):

- $\textsc{Ins}(a, p)$: 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 [25], 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 [1] 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 $e_1 = \text{do}(\textsc{Read}, w_1 \triangleq a_1^0 \ldots a_1^{m-1})$ and $e_2 = \text{do}(\textsc{Read}, w_2 \triangleq a_2^0 \ldots a_2^{n-1})$ ($a_i^j \in \text{elems}(A)$), 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 [1].

▶ **Definition 1** (Weak List Specification $\mathcal{A}_{\text{weak}}$ [1])**.** 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 $w = a_0 \ldots a_{n-1}$, 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:
   $$\forall a.\, a \in w \iff \left(\text{do}(\textsc{Ins}(a, \_), \_) \le_{\text{vis}} e\right) \land \neg\left(\text{do}(\textsc{Del}(a, \_), \_) \le_{\text{vis}} e\right).$$
   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) [6]: 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 is given in Figure A.1 [6, 11, 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 [28, 1]. 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.

An ***operation*** op of type Op (Algorithm B.1) 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 op "knows"; and *4)* $sctx$ is the operation context at the server denoting the operations that have been executed before op at the server. Operations are *totally ordered* by their $sctx$ components: $op \prec_s op' \iff op'.sctx = \emptyset \lor op.oid \in op'.sctx$ (CJupiter will not compare two operations both with $sctx = \emptyset$).

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 $Vertex$ (Algorithm B.2) is a pair $v = (oids, edges)$, where $oids \in 2^{\mathbb{N}_0 \times \mathbb{N}_0}$ is the set of operations (represented by their identifies) that have been executed, and $edges$ is an *ordered* set of edges of type $Edge$ (Algorithm B.3) 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

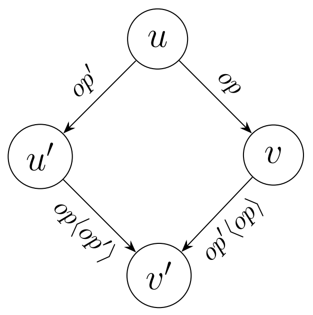


Figure 3 Illustration of an OT of two operations $op, op' \in Op$ in both the n-ary ordered state space of CJupiter and the 2D state space of Jupiter: $(op\langle op'\rangle, op'\langle op\rangle) = OT(op, op')$.

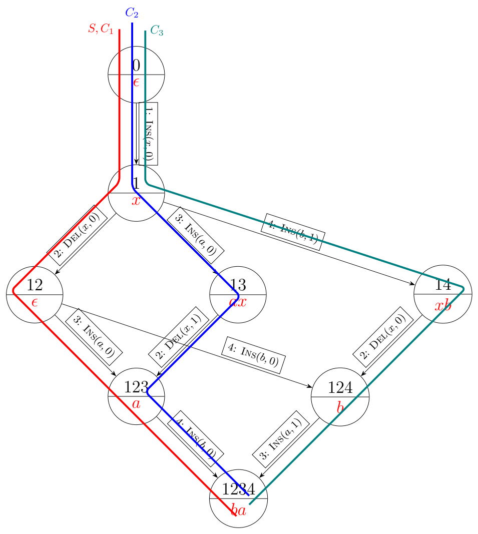


Figure 4 The same final n-ary ordered state space (thus for CSS$_s$ and each CSS$_{c_i}$) constructed by CJupiter for each replica under the schedule of Figure 1. Each replica behavior corresponds to a path going through this state space.

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 Lemma 5.

3.2 The CJupiter Protocol

Each replica in CJupiter maintains an n-ary ordered state space S of type *CStateSpace* (Algorithm B.4) and keeps the most recent vertex *cur* (initially $(\emptyset, \emptyset)$) of S. Following [28], we describe CJupiter in three parts.

Local Processing Part (Do of Algorithm B.5). 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 op "knows";
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 (Receive of Algorithm B.6). 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 $op.oid$;

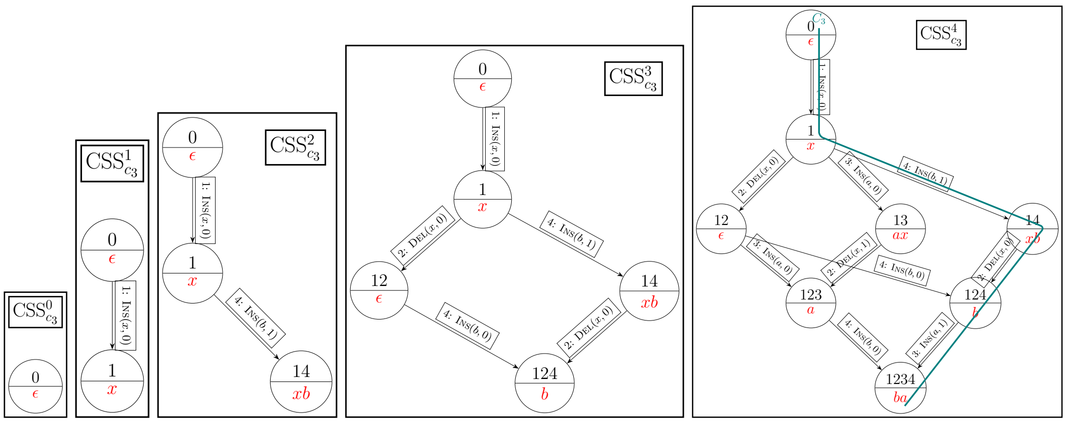


Figure 5 Illustration of client c_3 in CJupiter under the schedule of Figure 1. Its behavoir is indicated by the path in $\text{CSS}_{c_3}^4$.

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 (Receive of Algorithm B.5). 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.

OTs in CJupiter (xForm of Algorithm B.4). The procedure $S.\text{xForm}(op : Op)$ 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., $u.oids = op.ctx$ [3], 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 $\text{CSS}_{c_i}^k$ (resp. CSS_s^k). 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. (The illustration of other replicas is shown in Figures 4 and B.1; see

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

Section 3.3.) 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.

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 $\text{CSS}^1_{c_3}$ via an edge from v_1 labeled with $o_4\{o_1\}$. Then, $o_4\{o_1\}$ is propagated to the server.

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. By xForm, $o_2\{o_1\}$ is transformed with $o_4\{o_1\}$: $OT\Big(o_2\{o_1\} = \textsc{Del}(x, 0), o_4\{o_1\} = \textsc{Ins}(b, 1)\Big) = \Big(o_2\{o_1, o_4\} = \textsc{Del}(x, 0), o_4\{o_1, o_2\} = \textsc{Ins}(b, 0)\Big)$. As a result, v_{124} is created and is linked to v_{12} and v_{14} 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. 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 $o_3\{o_1, o_2\}$ from v_{12}. Specifically, $o_3\{o_1\}$ is first transformed with $o_2\{o_1\}$: $OT\Big(o_3\{o_1\} = \textsc{Ins}(a, 0), o_2\{o_1\} = \textsc{Del}(x, 0)\Big) = \Big(o_3\{o_1, o_2\} = \textsc{Ins}(a, 0), o_2\{o_1, o_3\} = \textsc{Del}(x, 1)\Big)$. 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, $o_3\{o_1, o_2\}$ is transformed with $o_4\{o_1, o_2\}$, 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 processed in total order ‘$\prec_s$’. For any vertex v in CSS_s, the path along the **first** edges from v to the final vertex of CSS_s is either empty (if v is the final vertex of CSS_s) or consists of the operations of $OP \setminus v$ in total order ‘$\prec_s$’, where*

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

▶ **Example 6** (CJupiter's “First” Rule). Consider CSS_s at the server shown in Figure 4 under the schedule of Figure 1. Take $OP = \langle o_1, o_2, o_3, o_4 \rangle$ in Lemma 5 (we mix operations of types $\mathcal{O}$ and Op). 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 [28].

▶ **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 and o_3 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. Each replica behavior corresponds to a path going through this state space.

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 [28], a *multi-client* description of Jupiter first proposed in [16]. Consider a client/server system with n clients. Jupiter [28] maintains $2n$ *2D state spaces*, 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 (see Appendix D for details): 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 (instead of following 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 also describe Jupiter in three parts. We omit the details that are in common with and have been explained in CJupiter.

Local Processing Part (Do of Algorithm D.5). 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.

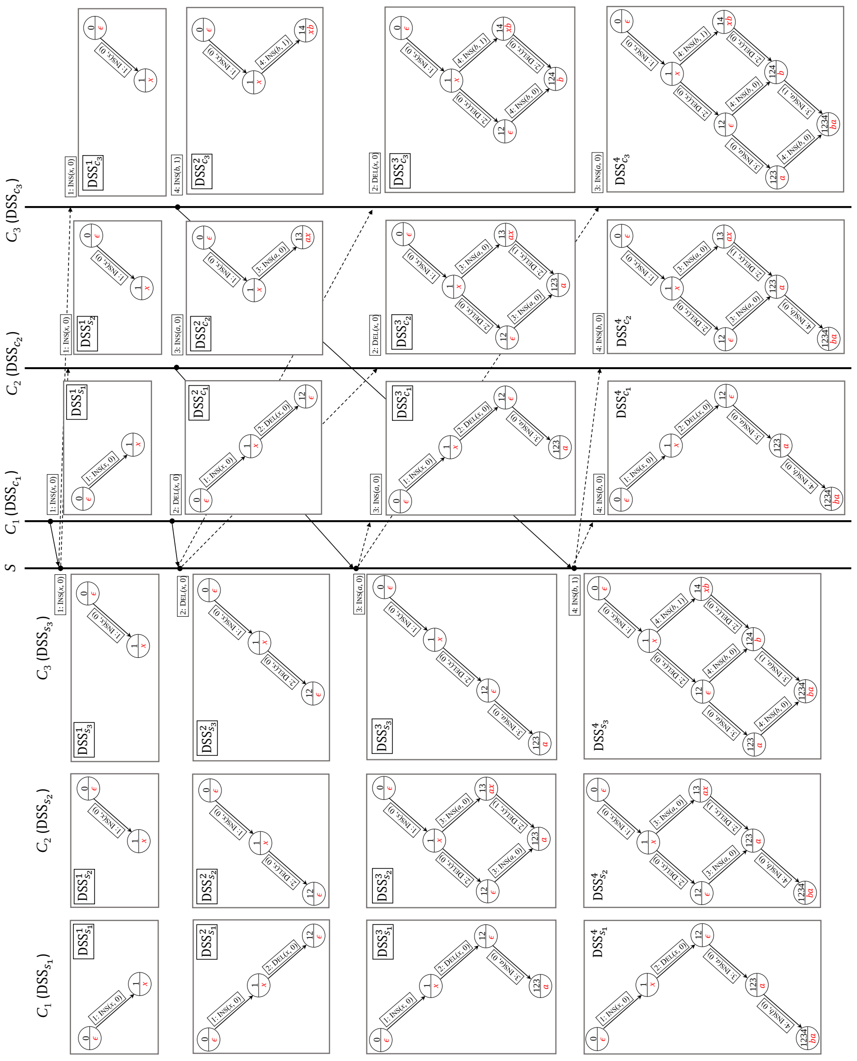


Figure 6 (Rotated) illustration of Jupiter [28] under the schedule of Figure 1.

Server Processing Part (Receive of Algorithm D.6). 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 xForm$(op, GLOBAL)$ (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 (Receive of Algorithm D.5). 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 xForm$(op, LOCAL)$ (see below), and applies op' locally.

OTs in Jupiter (xForm of Algorithm D.4). In the procedure xForm$(op : Op, d : LG = \{LOCAL, GLOBAL\})$ 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 Jupiter under the schedule of Figure 1. We emphasize three differences between CJupiter and Jupiter, by comparing Figures 4 and 5 with Figure 6. First, CJupiter maintains only a single n-ary ordered data structure CSS_s at the server (Figure 4) instead of multiple 2D state spaces DSS_{s_i} as in Jupiter (Figure 6). Second, each vertex in the n-ary ordered state space of CJupiter (Figure 4) 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; see $\text{CSS}_{c_3}^4$ of Figure 5 and $\text{DSS}_{c_3}^4$ of Figure 6.

▶ Remark. The optimization of Jupiter at clients can be easily integrated into CJupiter by letting its server propagate the transformed operations. Thus, in implementation, we should take advantage of both the server part of CJupiter and the client part of Jupiter.

4.2 The Servers Established Equivalent

According to [28], 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 [4] of all 2D state spaces DSS_{s_i} maintained at the server for each client c_i in Jupiter; see CSS_s of Figure 4 and DSS_{s_i}'s of Figure 6 for an example. The equivalence of servers are thus established.

▶ **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*

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

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

▶ **Theorem 15** (Equivalence of Servers). *Under the same schedule, the behaviors (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 $\text{DSS}_{c_i}^k$ of Jupiter is a part (i.e., subgraph) of $\text{CSS}_{c_i}^k$ 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*

$$DSS_{c_i}^k \subseteq CSS_{c_i}^k, \quad 1 \le i \le n,\ k \ge 1. \tag{$\star$}$$

▶ **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.*

[4] 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.

5 CJupiter Satisfies the Weak List Specification

The following theorem, together with Theorem 12, solves the conjecture of Attiya et al. [1].

▶ **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 $A = (H, \text{vis})$ with $\text{vis} = \xrightarrow{\text{hb}_\alpha}$ (Section 2.1). We then prove the conditions of $\mathcal{A}_{\mathsf{weak}}$ (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 $A = (H, \text{vis})$ belongs to $\mathcal{A}_{\mathsf{weak}}$, 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 $a, b \in \text{elems}(A)$, $a \xrightarrow{\text{lo}} b$ 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 $e = \text{do}(o, w) \in H$; *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.*

Proof. Due to the specific structure of OTs (Figure 3), all the transitions associated with the same operation are "parallel" in n-ary ordered state spaces. They cannot be in the same path. ◀

Proof. (For Lemma 21.) "$\Longleftarrow$" *(if):* Suppose by contradiction that $a \xrightarrow{\text{lo}} a$ for some $a \in \text{elems}(H)$. According to Lemma 22, a list state w contains no duplicate elements. Therefore, there exist two list states such that for some element b, $a \xrightarrow{\text{lo}} b$ (namely, a precedes b) in one state and $b \xrightarrow{\text{lo}} a$ (namely, b precedes a) in the other. However, this contradicts the assumption that all list states are pairwise compatible.

"$\Longrightarrow$" *(only if):* Suppose by contradiction that two list states w_1 and w_2 are incompatible. That is, they have two common elements a and b such that a precedes b in, say, w_1 and b precedes a in w_2. Thus, both $a \xrightarrow{\text{lo}} b$ and $b \xrightarrow{\text{lo}} a$ hold. Since lo is transitive on w_1 (and w_2), we have $a \xrightarrow{\text{lo}} a$, contradicting the assumption that lo is irreflexive. ◀

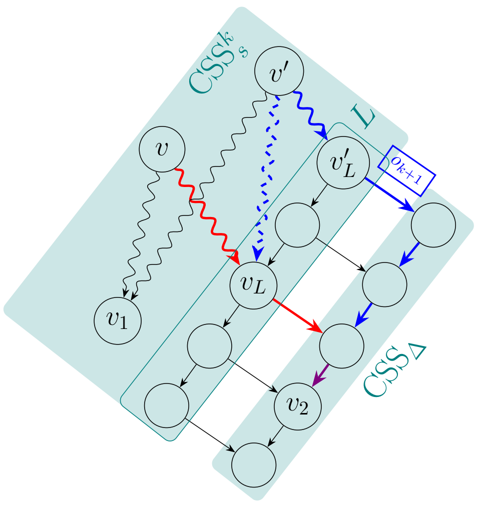


Figure 7 Illustration of proof for Lemma 23: vertices v and v' are two incomparable common ancestors of v_1 and $v'' = v_L \triangleq \min\{v_L, v'_L\}$ in CSS_s^k.

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*, every pair of vertices in the n-ary ordered state space CSS_s has a unique LCA (Lowest Common Ancestor).* [5]

Proof. By mathematical induction on the operation sequence $O = \langle op_1, op_2, \cdots, op_m \rangle$ ($op_i \in Op$) processed in total order '$\prec_s$' at the server.

Base Case. Initially, the n-ary ordered state space CSS_s^0 at the server contains only the single initial vertex $v_0 = (\emptyset, \emptyset)$. The lemma obviously holds.

Inductive Hypothesis. Suppose that the server has processed k operations and that every pair of vertices in the n-ary ordered state space CSS_s^k has a unique LCA.

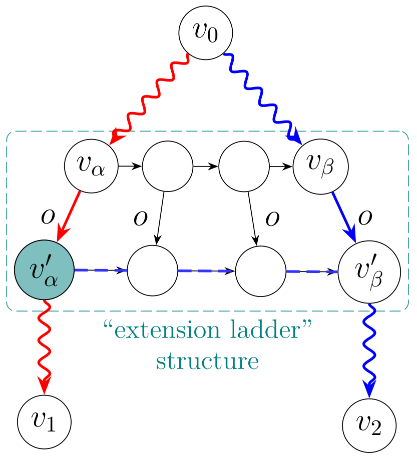
Inductive Step. The server has processed the $(k+1)$-*st* operation op_{k+1}. We shall prove that every pair of vertices in the n-ary ordered state space CSS_s^{k+1} has a unique LCA. Let

$$\text{CSS}_\Delta \triangleq \text{CSS}_s^{k+1} \setminus \text{CSS}_s^k$$

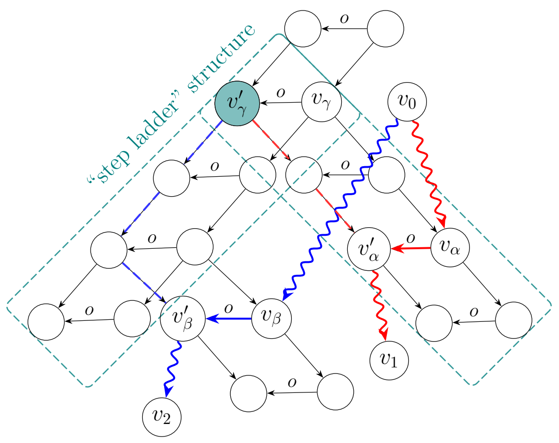
be the extra part of CSS_s^{k+1} obtained by transforming op_{k+1} with some operation sequence L in CSS_s^k (Figure 7). We need to verify that *1*) every pair of vertices in CSS_Δ has a unique LCA; *and 2*) every pair of vertices consisting of one vertex in CSS_s^k and the other in CSS_Δ has a unique LCA.

The former claim obviously holds because all vertices in CSS_Δ are in a path. We prove the latter by contradiction. Let v_1 be any vertex in CSS_s^k and v_2 any vertex in CSS_Δ

[5] The LCA of two vertices v_1 and v_2 in the n-ary ordered state space CSS_s, which is a directed acyclic graph, is the lowest (i.e., deepest) vertex that has both v_1 and v_2 as descendants.



(a) CASE 2.1: $v_\alpha \xrightarrow{o} v'_\alpha$ and $v_\beta \xrightarrow{o} v'_\beta$ are in the same "extension ladder".



(b) CASE 2.2: $v_\alpha \xrightarrow{o} v'_\alpha$ and $v_\beta \xrightarrow{o} v'_\beta$ are in a "step ladder".

Figure 8 Illustrations of CASE 2 of the proof for Lemma 24: v_1 and v_2 are not in the same path from $v_0 = \text{LCA}(v_1, v_2)$.

(Figure 7). Clearly, the initial vertex $v_0 = (\emptyset, \emptyset)$ is a common ancestor of v_1 and v_2. Suppose by contradiction that there are two LCAs, denoted v and v', of v_1 and v_2 in CSS_s^k (they cannot be in CSS_Δ).

Note that any path from v or v' (or generally, from any vertex in CSS_s^k) to v_2 passes through some vertex in L. Let v_L (resp. v'_L) be the last vertex in L in the path from v (resp. v') to v_2. Let $v'' = \min\{v_L, v'_L\}$ be the second vertex of v_L and v'_L along L. Then, v and v' are two incomparable common ancestors of v_1 and v'' (i.e., v_L in this example) that are both in CSS_s^k. This, however, contradicts the inductive hypothesis. ◀

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 $P_{v_0 \rightsquigarrow v_2}$.*

Proof. We consider whether v_1 and v_2 are in the same path from $v_0 = \text{LCA}(v_1, v_2)$ or not.

CASE *1: v_1 and v_2 are in the same path from $v_0 = \text{LCA}(v_1, v_2)$.* This lemma obviously holds because either $O_{v_0 \rightsquigarrow v_1}$ or $O_{v_0 \rightsquigarrow v_2}$ is empty.

CASE *2: v_1 and v_2 are not in the same path from $v_0 = \text{LCA}(v_1, v_2)$.* In this case, we prove this lemma by contradiction. Suppose that $o \in O_{v_0 \rightsquigarrow v_1} \cap O_{v_0 \rightsquigarrow v_2}$, where o can be either original or transformed (identified by its *oid*). As illustrated in Figure 8, the paths $P_{v_0 \rightsquigarrow v_1}$ and $P_{v_0 \rightsquigarrow v_2}$ are: $P_{v_0 \rightsquigarrow v_1} = P_{v_0 \rightsquigarrow v_\alpha \xrightarrow{o} v'_\alpha \rightsquigarrow v_1}$ and $P_{v_0 \rightsquigarrow v_2} = P_{v_0 \rightsquigarrow v_\beta \xrightarrow{o} v'_\beta \rightsquigarrow v_2}$. In the following, we derive a contradiction that v_0 is not the unique LCA of v_1 and v_2. We consider two cases according to how the edges $v_\alpha \xrightarrow{o} v'_\alpha$ and $v_\beta \xrightarrow{o} v'_\beta$ are related via OTs in CSS_s.

CASE *2.1: $v_\alpha \xrightarrow{o} v'_\alpha$ and $v_\beta \xrightarrow{o} v'_\beta$ are in the same "extension ladder" structure of OTs.* Without loss of generality, we assume that v'_β is reachable from v'_α; as illustrated in Figure 8a. In this case, v'_α is a *lower* common ancestor of v_1 and v_2 than v_0.

CASE *2.2: $v_\alpha \xrightarrow{o} v'_\alpha$ and $v_\beta \xrightarrow{o} v'_\beta$ are in a "step ladder" structure of OTs.* Because all the edges labeled with the same operation o are constructed directly or indirectly from the

OTs involving the original form of o, there exists some edge $v_\gamma \xrightarrow{o} v'_\gamma$ that is in the same "extension ladder" with $v_\alpha \xrightarrow{o} v'_\alpha$ as well as with $v_\beta \xrightarrow{o} v'_\beta$; as illustrated in Figure 8b. In this case, v'_γ is a common ancestor of v_1 and v_2 *other than* v_0. ◀

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.*

Proof. We prove a stronger statement that *every pair of vertices consisting of one vertex in $O_{v_0 \rightsquigarrow v_1}$ and the other in $O_{v_0 \rightsquigarrow v_2}$ are compatible*, by mathematical induction on the length l of the path $P_{v_0 \rightsquigarrow v_2}$. To this end, we first claim that

▶ **Claim** (One-step Compatibility). *Suppose that vertices v and v' are compatible. Let v'' be the next vertex of v' along the edge labeled with operation o which does not correspond to any element of the list in vertex v. Then, v and v'' are compatible.*

(Proof for this claim.) Let $C(v, v')$ be the set of common elements of lists in vertices v and v' and $C(v, v'')$ in vertices v and v''. Since o does not correspond to any element of the list in vertex v, $C(v, v'')$ is a subset of $C(v, v')$. Furthermore, the total ordering of elements in $C(v, v'')$ is consistent with that in $C(v, v')$. ■

Base Case. $l = 0$. $P_{v_0 \rightsquigarrow v_2}$ contains only the vertex v_0. We shall prove that v_0 is compatible with every vertex along $P_{v_0 \rightsquigarrow v_1}$. This can be done by mathematical induction on the length of $P_{v_0 \rightsquigarrow v_1}$ with the claim above and the fact that $P_{v_0 \rightsquigarrow v_1}$ is a simple path.

Inductive Hypothesis. Suppose that this lemma holds when the length of $P_{v_0 \rightsquigarrow v_2}$ is $l \geq 1$.

Inductive Step. We shall prove that the $(l+1)$-*st* vertex, denoted v_{l+1}, of $P_{v_0 \rightsquigarrow v_2}$ is compatible with every vertex along $P_{v_0 \rightsquigarrow v_1}$. This can be done by mathematical induction on the length of $P_{v_0 \rightsquigarrow v_1}$ with the claim above, the fact that $P_{v_0 \rightsquigarrow v_2}$ is a simple path (for v_0 and v_{l+1} being compatible), and the fact that $P_{v_0 \rightsquigarrow v_1}$ and $P_{v_0 \rightsquigarrow v_2}$ are disjoint. ◀

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. ◀

6 Distributed Jupiter Protocol

By orthogonally integrating the n-ary ordered state space with a distributed scheme to totally order operations, we further extend CJupiter to a distributed setting.

Recall that the construction of n-ary ordered state space at each replica is determined by the schedule of operations, namely the causal and total orderings on operations. Being distributed, the DJupiter (Distributed Jupiter) protocol relies on an atomic broadcast [5, 3] to totally order operations. Taking into account the requirement of preserving causality, DJupiter assumes a causal atomic broadcast service [3, 1] and behaves as CJupiter with two exceptions:

1. In the case of a replica generating an operation o, the replica finally broadcasts o to all replicas (including itself), utilizing the causal atomic broadcast service.
2. Each replica receives and processes operations in the order enforced by the causal atomic broadcast service, ignoring the operations it generates.

DJupiter is a *simulation* of CJupiter, by replacing the central server in CJupiter with an atomic broadcast service. Thus, it also satisfies the weak list specification $\mathcal{A}_{\text{weak}}$.

▶ **Proposition 27** (Simulation). *Let $\mathcal{R} = \{R_1, \ldots, R_n, S\}$ be the* CJupiter *protocol running on a client/server system with n clients $R_1, \ldots, R_n$ and a server S. Let $\mathcal{R}' = \{R'_1, \ldots, R'_n\}$ be the* DJupiter *protocol running on a distributed system with n replicas. Then $\mathcal{R}'$ is a simulation of $\mathcal{R}$ in the sense that for any execution α' of $\mathcal{R}'$, there exists an execution α of $\mathcal{R}$ such that*

$$\alpha|^{do}_{R_i} = \alpha'|^{do}_{R'_i}, \quad 1 \le i \le n.$$

Proof. Given an execution α' of $\mathcal{R}'$, we construct an execution α of $\mathcal{R}$ satisfying $\alpha|^{\text{do}}_{R_i} = \alpha'|^{\text{do}}_{R'_i}$ as follows.

Set $R_i = R'_i$, for $i = 1 \ldots n$. Execution α differs from α' in that:

- Instead of broadcasting a locally generated operation o, a replica sends o to the server S.
- The server S receives operations in the total order enforced by the causal atomic broadcast service in α'. Furthermore, the server S receives an operation o, say, from R_i, at some moment between o is broadcast by R'_i and o is received by the first replica in α'.
- The server S redirects these operations to all replicas in FIFO order.
- Each replica R_i receives operations at the moment R'_i does in α', and ignores the operations it generates.

◀

7 Related Work

Convergence is the main property for implementing a highly-available replicated list object [6, 28]. Since 1989 [6], a number of OT [6]-based protocols have been proposed. These protocols can be classified according to whether they rely on a total order on operations [28]. Various protocols like Jupiter [16, 28] establish a total order via a central server, a sequencer, or a distributed timestamping scheme [7, 27, 21, 14, 26]. By contrast, protocols like adOPTed [18] rely only on a partial (causal) order on operations [6, 17, 24, 23, 22].

In 2016, Attiya et al. [1] propose the strong/weak list specification of a replicated list object. They prove that the existing CRDT (Conflict-free Replicated Data Types) [20]-based RGA protocol [19] satisfies the strong list specification, and *conjecture* that the well-known OT-based Jupiter protocol [16, 28] satisfies the weak list specification.

The OT-based protocols typically use data structures like 1D buffer [21], 2D state space [16, 28], or N-dimensional interaction model [18] 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.

8 Conclusion and Future Work

We prove that the well-known Jupiter protocol [16, 28] satisfies the weak list specification [1], thus solving the conjecture recently proposed by Attiya et al. [1]. 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, based on which we study whether the distributed adOPTed protocol [18] satisfies the strong/weak list specification.

References

1 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.

2 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.

3 Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley & Sons, NJ, USA, 2004.

4 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.

5 Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, December 2004.

6 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.

7 The Apache Software Foundation. Apache wave. `https://incubator.apache.org/wave/`, 2012. Accessed: August 2017.

8 The Apache Wave Foundation. *Apache Wave (incubating) Protocol Documentation (Release 0.4)*, August 22, 2015.

9 Google. Google docs. `https://docs.google.com`. Accessed: January 2018.

10 Google. What's different about the new google docs: Making collaboration fast. `https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html`, 2010. Accessed: May 2018.

11 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.

12 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

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

14 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. IEEE Computer Society, 2004.

15 Codoxware Pte Ltd. Codoxword. `http://www.codoxware.com/`, 2014. Accessed: August 2017.

16 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.

17 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.

18 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.

19 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.

20 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.

21 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.

22 Chengzheng Sun. Undo as concurrent inverse in group editors. *ACM Trans. Comput.-Hum. Interact.*, 9(4):309–361, December 2002.

23 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.

24 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.

25 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.

26 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.

27 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.

28 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.

A The OT System

According to Section 2.1, we represent the replica state in an OT system (including both Jupiter and CJupiter) as a sequence of operations $\langle o_1, o_2, \cdots, o_m \rangle$ (where, $o_i \in \mathcal{O}$).

The function

$$\textsc{Apply} : \Sigma \times \mathcal{O} \rightarrow \Sigma \times \mathit{Val}$$

applies an operation o to a state σ, returning a new state $\sigma \circ o$ and the list content produced by performing $\sigma \circ o$ on the initial list.

Figure A.1 shows the OT functions satisfying CP1 for a replicated list object [6, 11]. Operations Ins and Del have been extended with an extra parameter pr for “priority” [11]. It helps to resolve the conflicts when two concurrent Ins operations are intended to insert different elements at the same position. We assume that the operations generated by the replica with a smaller identifier have a higher priority. When a conflict occurs, the insertion position of the Ins operation with a higher priority will be shifted.

We highlight one property of OTs that when transformed in both Jupiter and CJupiter, the type and effect of an insertion (resp. a deletion) $\textsc{Ins}(a, p)$ (resp. $\textsc{Del}(a, p)$) 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).

Figure 3 illustrates an OT of two operations $op, op' \in Op$ in both the n-ary ordered state space of CJupiter and the 2D state space of Jupiter:

$$(op\langle op' \rangle, op'\langle op \rangle) = OT(op, op').$$

Algorithm A.1 lists the constants used in Jupiter and/or CJupiter.

Algorithm A.1 Constants.

```
    ▷ for both Jupiter and CJupiter
1:  $SID = 0$
2:  $CID = \{1 \cdots n\}$
3:  $SEQ = \mathbb{N}_0$

4:  Enum $LG$ $\{LOCAL = 0, GLOBAL = 1\}$                ▷ for Jupiter
5:  Enum $Ord$ $\{LT = -1, EQ = 0, GT = 1\}$            ▷ for CJupiter
```

$$OT\Big(\textsc{Ins}(a_1, p_1, pr_1), \textsc{Ins}(a_2, p_2, pr_2)\Big) = \begin{cases} \textsc{Ins}(a_1, p_1, pr_1) & p_1 < p_2 \\ \textsc{Ins}(a_1, p_1 + 1, pr_1) & p_1 > p_2 \\ \text{NOP} & p_1 = p_2 \wedge a_1 = a_2 \\ \textsc{Ins}(a_1, p_1 + 1, pr_1) & p_1 = p_2 \wedge a_1 \neq a_2 \wedge pr_1 > pr_2 \\ \textsc{Ins}(a_1, p_1, pr_1) & p_1 = p_2 \wedge a_1 \neq a_2 \wedge pr_1 \leq pr_2 \end{cases}$$

$$OT\Big(\textsc{Ins}(a_1, p_1, pr_1), \textsc{Del}(\_, p_2, pr_2)\Big) = \begin{cases} \textsc{Ins}(a_1, p_1, pr_1) & p_1 \leq p_2 \\ \textsc{Ins}(a_1, p_1 - 1, pr_1) & p_1 > p_2 \end{cases}$$

$$OT\Big(\textsc{Del}(\_, p_1, pr_1), \textsc{Ins}(a_2, p_2, pr_2)\Big) = \begin{cases} \textsc{Del}(\_, p_1, pr_1) & p_1 < p_2 \\ \textsc{Del}(\_, p_1 + 1, pr_1) & p_1 \geq p_2 \end{cases}$$

$$OT\Big(\textsc{Del}(\_, p_1, pr_1), \textsc{Del}(\_, p_2, pr_2)\Big) = \begin{cases} \textsc{Del}(\_, p_1, pr_1) & p_1 < p_2 \\ \textsc{Del}(\_, p_1 - 1, pr_1) & p_1 > p_2 \\ \text{NOP} & p_1 = p_2 \end{cases}$$

Figure A.1 The OT functions satisfying CP1 for a replicated list object [6, 11]. The parameter "*pr*" means "priority" which helps to resolve the conflicts when two concurrent Ins operations are intended to insert different elements at the same position. The elements to be deleted in Del operations are irrelevant and are thus represented by '_'s.

B The CJupiter Protocol

Algorithm B.1 Operation in CJupiter.

```
1: Class $Op$ begin
2:     Var $o : \mathcal{O}$
3:     Var $oid : CID \times SEQ$
4:     Var $ctx : 2^{CID \times SEQ} = \emptyset$
5:     Var $sctx : 2^{CID \times SEQ} = \emptyset$

       ▷ Precondition: $op.sctx \neq \emptyset \mid\mid op'.sctx \neq \emptyset$
6:     procedure COMPARE($op : Op, op' : Op$) : $Ord$
7:         if $op.oid == op'.oid$ then
8:             return $EQ$
9:         else if $op'.sctx = \emptyset \mid\mid op.oid \in op'.sctx$ then
10:            return $LT$                                    ▷ $op \prec_s op'$
11:        else
12:            return $GT$                                    ▷ $op' \prec_s op$
13:        end if
14:    end procedure

15:    procedure OT($op : Op, op' : Op$) : $(Op, Op)$
16:        $(o, o') \leftarrow \mathrm{OT}(op.o, op.o')$                  ▷ call OT on $\mathcal{O}$
17:        $Op\ op\langle op' \rangle = \text{ new } Op(o, op.oid, op.ctx \cup \{op'.oid\}, op.sctx)$
18:        $Op\ op'\langle op \rangle = \text{ new } Op(o', op'.oid, op'.ctx \cup \{op.oid\}, op'.sctx)$
19:        return $(op\langle op' \rangle, op'\langle op \rangle)$
20:    end procedure
21: end                                                       ▷ Class $Op$
```

Algorithm B.2 Vertex in the n-ary ordered state space.

```
Class Vertex begin
    Var oids : $2^{CID \times SEQ} = \emptyset$
    Var edges : SortedSet$\langle Edge \rangle = \emptyset$

    procedure FIRSTCHILDOP() : Op
        return edges.FIRST().op
    end procedure

    procedure FIRSTCHILDVERTEX() : Vertex
        return edges.FIRST().v
    end procedure
end                                                    ▷ Class Vertex
```

Algorithm B.3 Edge in the n-ary ordered state space.

```
Class Edge begin
    Var op : $Op = \Lambda$
    Var v : $Vertex = \Lambda$

    procedure COMPARE($e : Edge, e' : Edge$) : Ord
        return COMPARE($e.op, e'.op$)
    end procedure
end                                                    ▷ Class Edge
```

Algorithm B.4 The n-ary ordered state space.

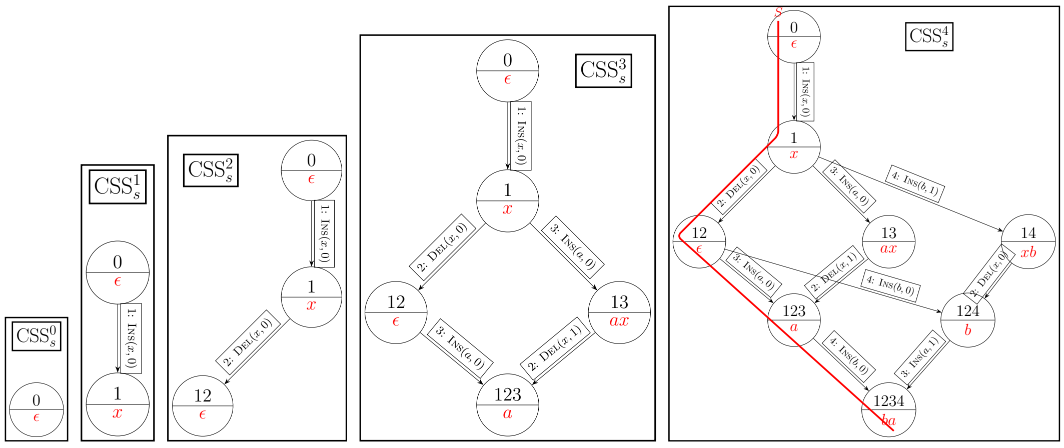
```
1: Class CStateSpace begin
2:     Var $cur : Vertex$= new $Vertex()$

3:     procedure xForm($op : Op$) : $Op$
4:         Vertex $u \leftarrow$ Locate($op$)
5:         Vertex $v \leftarrow$ new $Vertex(u.oids \cup \{op.oid\}, \emptyset)$
6:         while $u \neq cur$ do                                   ▷ See Figure 3
7:             Vertex $u' \leftarrow u$.firstChildVertex()
8:             Op $op' \leftarrow u$.firstChildOp()
9:             $(op\langle op' \rangle, op'\langle op \rangle) \leftarrow \mathrm{OT}(op, op')$
10:            Vertex $v' \leftarrow$ new $Vertex(v.oids \cup \{op'.oid\}, \emptyset)$
11:            Link($v, v', op'\langle op \rangle$)
12:            Link($u, v, op$)
13:            $u \leftarrow u'$
14:            $v \leftarrow v'$
15:            $op \leftarrow op\langle op' \rangle$
16:        end while
17:        Link($u, v, op$)
18:        $cur \leftarrow v$
19:        return $op$
20:    end procedure

21:    procedure Locate($op : Op$) : $Vertex$
22:        return Vertex $v$ with $v.oids = op.ctx$
23:    end procedure

24:    procedure Link($u : Vertex, v : Vertex, op : Op$)
25:        Edge $e \leftarrow$ new $Edge(op, v)$
26:        $u.edges$.add($e$)
27:    end procedure
28: end                                                        ▷ Class CStateSpace
```

Algorithm B.5 Client in CJupiter.

```
Class $Client$ begin
    Var $cid$ : $CID$
    Var $seq$ : $SEQ = 0$
    Var $state$ : $\Sigma = \langle\rangle$                          ▷ a sequence of $o \in \mathcal{O}$
    Var $S$ : $CStateSpace$= new $CStateSpace()$

    procedure Do($o : \mathcal{O}$) : $Val$                           ▷ Local Processing
        $(state, val) \leftarrow$ Apply($state, o$)
        $seq \leftarrow seq + 1$
        $Op\ op \leftarrow$ new $Op(o, (cid, seq), S.cur.oids, \emptyset)$
        $Vertex\ v \leftarrow$ new $Vertex(S.cur.oids \cup \{op.oid\}, \emptyset)$
        Link($S.cur, v, op$)
        $S.cur \leftarrow v$
        Send($SID, op$)                                          ▷ send $op$ to the server
        return $val$
    end procedure

    procedure Receive($op : Op$)                                 ▷ Remote Processing
        $Op\ op' \leftarrow S.$xForm($op$)
        $state \leftarrow state \circ op'.o$
    end procedure
end                                                              ▷ Class $Client$
```

Algorithm B.6 Server in CJupiter.

```
Class $Server$ begin
    Var $state$ : $\Sigma = \langle\rangle$                          ▷ a sequence of $o \in \mathcal{O}$
    Var $cur\_oids$ : $2^{CID \times SEQ} = \emptyset$
    Var $S$ : $CStateSpace$= new $CStateSpace()$

    procedure Receive($op : Op$)                                 ▷ Server Processing
        $op.sctx \leftarrow cur\_oids$
        $cur\_oids \leftarrow cur\_oids \cup \{op.oid\}$
        $Op\ op' \leftarrow S.$xForm($op$)
        $state \leftarrow state \circ op'.o$
        for all $c \in CID \setminus \{op.oid.cid\}$ do
            Send($c, op$)                                        ▷ send $op$ (not $op'$) to client $c$
        end for
    end procedure
end                                                              ▷ Class $Server$
```


(a) Server s.

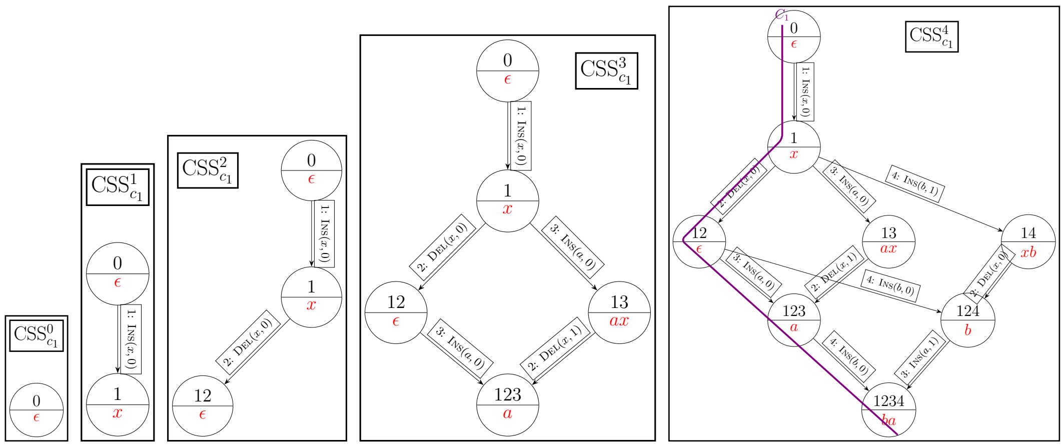
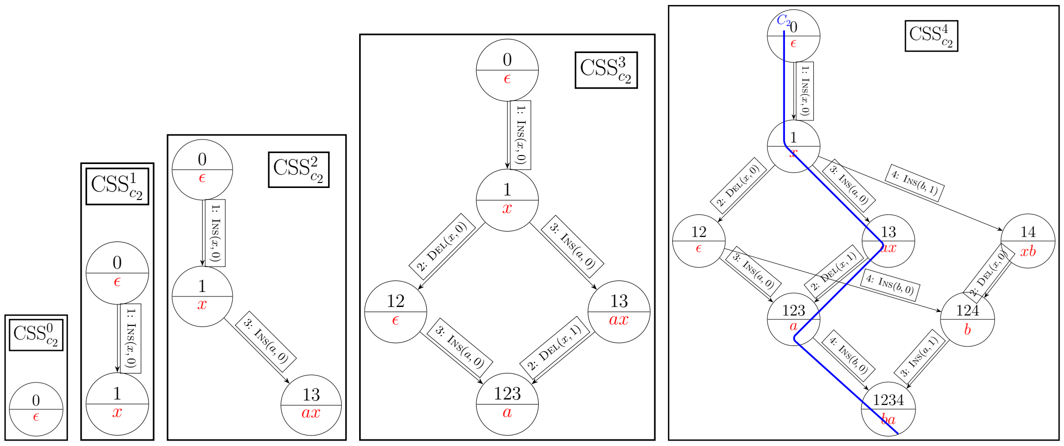

(b) Client c_1.

Figure B.1 Illustration of CJupiter under the schedule of Figure 1 (in the text). The replica behavoirs are indicated by the paths in the n-ary ordered state spaces. (To be continued)


(c) Client c_2.

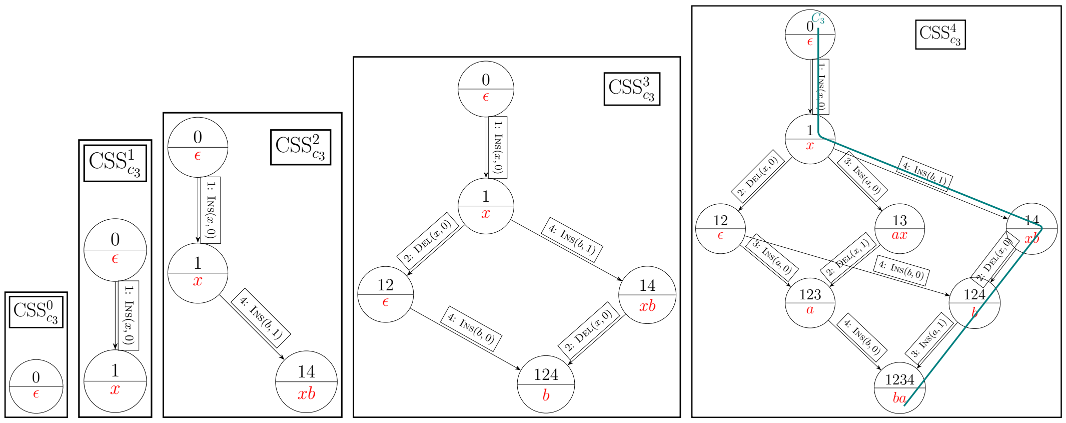

(d) Client c_3.

Figure B.1 (Continued.) Illustration of CJupiter under the schedule of Figure 1 (in the text). The replica behavoirs are indicated by the paths in the n-ary ordered state spaces.

C Proofs for Section 3: The CJupiter Protocol

C.1 Proof for Lemma 5 (CJupiter's "First" Rule)

Proof. By mathematical induction on the operation sequence O the server processes.

Base Case: $O = \langle\rangle$. CSS_s contains only the initial vertex $v_0 = (\emptyset, \emptyset)$ and the first edge from v_0 is empty.

Inductive Hypothesis: Suppose that the lemma holds for

$$O_k = \langle op_1, op_2, \ldots, op_k\rangle.$$

Inductive Step: Consider $O_{k+1} = \langle op_1, op_2, \ldots, op_k, op_{k+1}\rangle$. Suppose that the matching vertex of operation op_{k+1} is v (i.e., $v.oids = op_{k+1}.ctx$). We distinguish between v being the final vertex of CSS_s^k, denoted v_k, or not.

CASE 1: $v = v_k$. According to the procedure xFORM of CJupiter (Algorithm B.4), the state space CSS_s^{k+1} is obtained by extending CSS_s^k with a new edge from v_k labeled with op_{k+1}. Thus, each path consisting of first edges in CSS_s^k is extended by the edge labeled with op_{k+1}, meeting the second condition of the lemma in CSS_s^{k+1}. In addition, the first edge from the final vertex of CSS_s^{k+1} is empty, meeting the first condition.

CASE 2: $v \neq v_k$. According to the procedure xFORM of CJupiter (Algorithm B.4), the server transforms op_{k+1} with the operation sequence, denoted L_k, along the first edges from v to the final vertex v_k of CSS_s^k, obtaining the state space CSS_s^{k+1} with final vertex v_{k+1}. By inductive hypothesis, L_k consists of the operations in $O_k \setminus v$ in the total order '$\prec_s$'. To prove that the lemma holds for CSS_s^{k+1}, we need to check that (Figure C.1):

1. *It holds for old vertices in CSS_s^k.* Each path consisting of first edges from vertices in CSS_s^k is extended by the edge labeled with op_{k+1}, meeting the second condition of the lemma in CSS_s^{k+1}.
2. *It holds for new vertices in $CSS_s^{k+1} \setminus CSS_s^k$.* This is because these new vertices form a path along which the corresponding operation sequence is exactly L_k.

◀

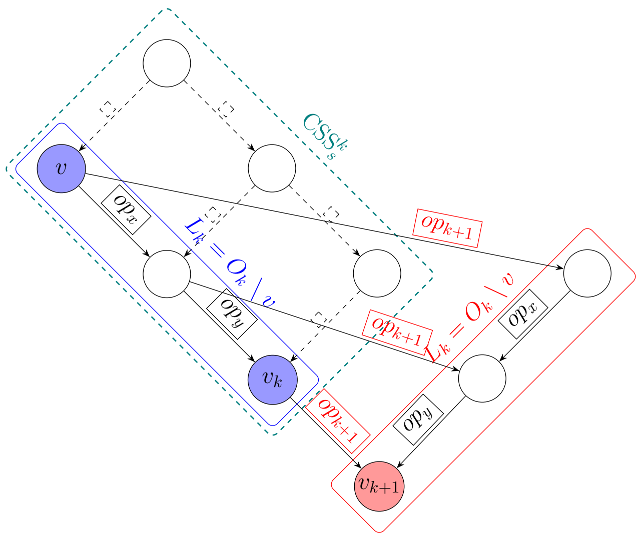


Figure C.1 Illustration of CASE 2 ($v \neq v_k$) of the proof for Lemma 5.

C.2 Proof for Lemma 7 (CJupiter's OT Sequence)

Proof. We show that if L is not empty, then

1. *All operations in L are totally ordered by '$\prec_s$' before op.* This holds because operation op is the last one in the total order '$\prec_s$'.
2. *All operations in L are concurrent by '$\parallel$' with op.* By contradiction. Suppose that some op' in L is not concurrent with op. Then it must be the case that $op' \to op$ and thus op' is not in L.
3. *L consists of all the operations satisfying 2) and 3) and all operations in L are totally ordered by '$\prec_s$'.* This is due to Lemma 5.

◀

C.3 Proof for Proposition 9 ($n+1 \Rightarrow 1$)

Proof. By mathematical induction on the number of operations in the schedule. Because all operations are serialized at the server, we proceed by mathematical induction on the operation sequence

$$O = \langle op_1, op_2, \ldots, op_m \rangle \; (op_i \in Op)$$

the server processes in total order '$\prec_s$'.

Base Case. $O = \langle op_1 \rangle$. There is only one operation in the schedule. When all replicas have eventually processed this operation, they obviously have the same n-ary ordered state space. Formally,

$$\text{CSS}_s^1 = \text{CSS}_{c_i}^1, \quad \forall 1 \le i \le n.$$

Inductive Hypothesis. $O = \langle op_1, op_2, \ldots, op_k \rangle$. Suppose that when all replicas have eventually processed all the k operations, they have the same n-ary ordered state space. Formally,

$$\text{CSS}_s^k = \text{CSS}_{c_i}^k, \quad \forall 1 \le i \le n.$$

Inductive Step. $O = \langle op_1, op_2, \ldots, op_{k+1} \rangle$. Suppose that the $(k+1)$-*st* operation op_{k+1} processed at the server is generated by client c_j. We shall prove that for any client c_i, when it has eventually processed all these $(k+1)$ operations, it has the same n-ary ordered state space as the server. Formally,

$$\text{CSS}_s^{k+1} = \text{CSS}_{c_i}^{k+1}, \quad \forall 1 \le i \le n.$$

In the following, we distinguish client c_j that generates op_{k+1} (more specifically, $op_{k+1}.o$ of type $\mathcal{O}$) from other clients.

CASE 1: $i \ne j$. The n-ary ordered state space CSS_s^{k+1} at the server is obtained by applying the $(k+1)$-*st* operation op_{k+1} to CSS_s^k, denoted by

$$\text{CSS}_s^{k+1} = op_{k+1} \otimes \text{CSS}_s^k.$$

Since the communication is FIFO and in CJupiter the original operation (i.e., op_{k+1} here) rather than the transformed one is propagated to clients by the server, the n-ary ordered state space $\text{CSS}_{c_i}^{k+1}$ at client c_i is obtained by applying the operation op_{k+1} to $\text{CSS}_{c_i}^k$, denoted by

$$\text{CSS}_{c_i}^{k+1} = op_{k+1} \otimes \text{CSS}_{c_i}^k.$$

By the inductive hypothesis,

$$\text{CSS}_s^k = \text{CSS}_{c_i}^k, \quad i \neq j.$$

Therefore, we have

$$\text{CSS}_s^{k+1} = \text{CSS}_{c_i}^{k+1}.$$

CASE 2: $i = j$. Now we consider client c_j that generates the operation op_{k+1}.

Let $\sigma_{k+1}^{c_j} \triangleq \langle op_1^{c_j}, op_2^{c_j}, \ldots, op_{k+1}^{c_j} \rangle$, [1] a permutation of $\sigma_{k+1}^s \triangleq O$ (i.e., $\langle op_1, op_2, \ldots, op_{k+1} \rangle$), be the operation sequence executed at client c_j. The operation op_{k+1} may not be the last one executed at client c_j. Instead, suppose op_{k+1} is the *l-th* $(1 \leq l \leq k+1)$ operation executed at client c_j, namely $op_l^{c_j} \equiv op_{k+1}$.

The operation $op_l^{c_j}$ splits the sequence $\sigma_{k+1}^{c_j}$ into three parts: the subsequence $\sigma_{1,l-1}^{c_j}$ consisting of the first $(l-1)$ operations, the subsequence $\sigma_{l,l}^{c_j}$ containing the operation $op_l^{c_j} \equiv op_{k+1}$ only, and the subsequence $\sigma_{l+1,k+1}^{c_j}$ consisting of the last $(k-l+1)$ operations. We formally denote this by

$$\sigma_{k+1}^{c_j} = \sigma_{1,l-1}^{c_j} \circ op_{k+1} \circ \sigma_{l+1,k+1}^{c_j}.$$

We remark that all operations in $\sigma_{l+1,k+1}^{c_j}$ are concurrent by '$\|$' with op_{k+1}, because they are generated by other clients than c_j before op_{k+1} reaches these clients and op_{k+1} is generated before they reach op_{k+1}'s local replica (i.e., c_j). Furthermore, due to the FIFO communication, the operations in $\sigma_{l+1,k+1}^{c_j}$ are totally ordered by '$\prec_s$'.

Let $\sigma_k^{c_j} \triangleq \langle op_1^{c_j}, op_2^{c_j}, \ldots, op_{l-1}^{c_j}, op_{l+1}^{c_j}, \ldots, op_{k+1}^{c_j} \rangle$ be the operation sequence obtained by deleting $op_l^{c_j}$ (i.e., op_{k+1}) from $\sigma_{k+1}^{c_j}$, namely

$$\sigma_k^{c_j} = \sigma_{1,l-1}^{c_j} \circ \sigma_{l+1,k+1}^{c_j}.$$

Thus, $\sigma_k^{c_j}$ is a permutation of $\sigma_k^s \triangleq \langle op_1, op_2, \ldots, op_k \rangle$.

In the following, we prove that the n-ary ordered state space $\text{CSS}_{c_j}^{k+1}$ at client c_j constructed by executing $\sigma_{k+1}^{c_j}$ in sequence, namely

$$\text{CSS}_{c_j}^{k+1} = \sigma_{k+1}^{c_j} \otimes \text{CSS}_{c_j}^0,$$

is the same with the n-ary ordered state space CSS_s^{k+1} at the server constructed by applying the $(k+1)$-*st* operation op_{k+1} to CSS_s^k, namely

$$\text{CSS}_s^{k+1} = op_{k+1} \otimes \text{CSS}_s^k.$$

By the inductive hypothesis, CSS_s^k would be the same with the n-ary ordered state space $\text{CSS}_{c_j}^k$ constructed at client c_j if it had processed $\sigma_k^{c_j}$ in sequence. Formally,

$$\text{CSS}_s^k = \text{CSS}_{c_j}^k \; (\triangleq \sigma_k^{c_j} \otimes \text{CSS}_{c_j}^0).$$

Therefore, it suffices to prove that the n-ary ordered state space $\text{CSS}_{c_j}^{k+1}$ at client c_j constructed by executing

$$\sigma_{k+1}^{c_j} = \sigma_{1,l-1}^{c_j} \circ op_{k+1} \circ \sigma_{l+1,k+1}^{c_j} \tag{1}$$

[1] We abuse the symbol 'σ' for representing states to denote operation sequences. This is reasonable because replica states are defined by the operations a replica has processed (Section 2.1).

in sequence would be the same with the n-ary ordered state space constructed at client c_j if it had processed

$$\sigma_k^{c_j} \circ op_{k+1} = \sigma_{1,l-1}^{c_j} \circ \sigma_{l+1,k+1}^{c_j} \circ op_{k+1} \tag{2}$$

in sequence.

We first consider the n-ary ordered state space obtained by applying op_{k+1} to $\text{CSS}_{c_j}^k$ (which is obtained after executing $\sigma_k^{c_j}$) at client c_j, corresponding to (2). The matching vertex of op_{k+1} is $\sigma_{1,l-1}^{c_j}$. According to Lemma 7 and the inductive hypothesis that CSS_s^k $= \text{CSS}_{c_j}^k$, the operation sequence L with which op_{k+1} transforms consists of exactly the (possibly transformed) operations in $\sigma_{l+1,k+1}^{c_j}$:

$$\begin{aligned} L: \; & op_{l+1}^{c_j}\{\sigma_{1,l-1}^{c_j}\},\; op_{l+2}^{c_j}\{\sigma_{1,l-1}^{c_j} \circ op_{l+1}^{c_j}\},\; \ldots, \\ & op_{l+3}^{c_j}\{\sigma_{1,l-1}^{c_j} \circ op_{l+1}^{c_j} \circ op_{l+2}^{c_j}\},\; op_{k+1}^{c_j}\{\sigma_{1,l-1}^{c_j} \circ op_{l+1}^{c_j} \circ \ldots \circ op_{k}^{c_j}\}. \end{aligned}$$

We now consider the construction of $\text{CSS}_{c_j}^{k+1}$ by executing $\sigma_{k+1}^{c_j}$ in three stages, corresponding to (1).

1. At the beginning, it grows as $\text{CSS}_{c_j}^k$ does when executing the common subsequence $\sigma_{1,l-1}^{c_j}$.
2. Next, the operation op_{k+1} is generated at client c_j. According to the local processing of CJupiter, the n-ary ordered state space grows by saving op_{k+1} at the final vertex (corresponding to) $\sigma_{1,l-1}^{c_j}$ along a new edge.
3. Then, the sequence $\sigma_{l+1,k+1}^{c_j}$ of operations (from the server) are processed at client c_j. Each operation in $\sigma_{l+1,k+1}^{c_j}$, when executed in sequence, not only "simulates" the growth of $\text{CSS}_{c_j}^k$, but also completes one step of the iterative operational transformations of op_{k+1} with the sequence L mentioned above when applying op_{k+1} to $\text{CSS}_{c_j}^k$. (This can be proved by mathematical induction.) We take as an example the case of the first operation $op_{l+1}^{c_j}$. After transforming with some subsequence of operations (which may be empty) in $\sigma_{1,l-1}^{c_j}$, operation $op_{l+1}^{c_j}$ is transformed as $op_{l+1}^{c_j}\{\sigma_{1,l-1}^{c_j}\}$. At that time, $op_{l+1}^{c_j}\{\sigma_{1,l-1}^{c_j}\}$ is then transformed with $op_{k+1}\{\sigma_{1,l-1}^{c_j}\}$, which is also performed when applying op_{k+1} to $\text{CSS}_{c_j}^k$:

$$\begin{aligned} & OT(op_{l+1}^{c_j}\{\sigma_{1,l-1}^{c_j}\}, op_{k+1}\{\sigma_{1,l-1}^{c_j}\}) \\ = & \big(op_{l+1}^{c_j}\{\sigma_{1,l-1}^{c_j} \circ op_{k+1}\}, op_{k+1}\{\sigma_{1,l-1}^{c_j} \circ op_{l+1}^{c_j}\}\big). \end{aligned}$$

   As it goes on, after executing $\sigma_{k+1}^{c_j}$ in sequence, we obtain an n-ary ordered state space same with that obtained by applying op_{k+1} to $\text{CSS}_{c_j}^k$.

◀

D The Jupiter Protocol

We review the Jupiter protocol in [28], a *multi-client* description of Jupiter first proposed in [16].

D.1 Data Structure: 2D State Space

For a client/server system with n clients, Jupiter maintains $2n$ 2D state spaces, each of which consists of a local dimension and a global dimension. We first define operations and vertices as follows.

▶ **Definition 28** (Operation)**.** Each operation op of type Op (Algorithm D.1) is a tuple $op = (o, oid, ctx)$, where

- o : the signature of type $\mathcal{O}$ described in Section 2.3;
- oid : a globally unique identifier which is a pair (cid, seq) consisting of the client id and a sequence number; *and*
- ctx : an *operation context* which is a set of operation identifiers, denoting the operations op "knows".

The OT functions of two operations $op, op' \in Op$,

$$OT : Op \times Op \to Op \times Op$$
$$(op\langle op'\rangle, op'\langle op\rangle) = OT(op, op'),$$

are defined based on those of operations $op.o, op'.o \in \mathcal{O}$, denoted $(o, o') = OT(op.o, op'.o)$, such that

$$op\langle op'\rangle = (o, op.oid, op.ctx \cup \{op'.oid\}),$$
$$op'\langle op\rangle = (o', op'.oid, op'.ctx \cup \{op.oid\}).$$

A 2D state space is a finite set of vertices.

▶ **Definition 29** (Vertex)**.** A vertex v of type $Vertex$ (Algorithm D.2) is a pair $v = (oids, edges)$, where

- $oids \in 2^{\mathbb{N}_0 \times \mathbb{N}_0}$ is the set of operations (represented by their identifies) that have been executed.
- $edges$ is an array of *two* (indexed by $LOCAL$ and $GLOBAL$) edges of type $Edge$ (Algorithm D.3) from v to two other vertices, labeled with operations. That is, each edge is a pair $(op : Op, v : Vertex)$.

For vertex u, we say that $u.edges[LOCAL].op$ is an operation from u along the *local* dimension/edge and $u.edges[GLOBAL].op$ along the *remote* dimension/edge. This is similar for the child vertices $u.edges[LOCAL].v$ and $u.edges[GLOBAL].v$ of u.

As with in an n-ary ordered state space, for each vertex v and each edge e from v in a 2D state space, it is required that

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

Algorithm D.1 Operation in Jupiter.

```
1:  Class $Op$ begin
2:      Var $o \,:\, \mathcal{O}$
3:      Var $oid \,:\, CID \times SEQ$
4:      Var $ctx \,:\, 2^{CID \times SEQ} = \emptyset$

5:      procedure $\mathrm{OT}(op : Op, op' : Op) \,:\, (Op, Op)$
6:          $(o, o') \leftarrow \mathrm{OT}(op.o, op.o')$                ▷ call OT on $\mathcal{O}$
7:          $Op\ op\langle op' \rangle =$ new $Op(o, op.oid, op.ctx \cup \{op'.oid\})$
8:          $Op\ op'\langle op \rangle =$ new $Op(o', op'.oid, op'.ctx \cup \{op.oid\})$
9:          return $(op\langle op' \rangle, op'\langle op \rangle)$
10:     end procedure
11: end                                                              ▷ Class $Op$
```

Algorithm D.2 Vertex in the 2D state space.

```
1:  Class $Vertex$ begin
2:      Var $oids \,:\, 2^{CID \times SEQ} = \emptyset$
3:      Var $edges \,:\, Edge[2] = \{[LOCAL] = [GLOBAL] = \Lambda\}$
4:  end                                                              ▷ Class $Vertex$
```

Algorithm D.3 Edge in the 2D state space.

```
1:  Class $Edge$ begin
2:      Var $op \,:\, Op = \Lambda$
3:      Var $v \,:\, Vertex = \Lambda$
4:  end                                                              ▷ Class $Edge$
```

Algorithm D.4 2D state space.

```
 1: Class $StateSpace2D$ begin
 2:     Var $cur$ : $Vertex$= new $Vertex()$

 3:     procedure xForm($op : Op, d : LG$) : $Op$
 4:         $Vertex$ $u \leftarrow$ Locate($op$)
 5:         $Vertex$ $v \leftarrow$ Add($op, 1 - d, u$)
 6:         while $u \neq cur$ do                                ▷ See Figure 3
 7:             $Vertex$ $u' \leftarrow u.edges[d].v$
 8:             $Op$ $op' \leftarrow u.edges[d].op$
 9:             $(op\langle op' \rangle, op'\langle op \rangle) \leftarrow$ OT($op, op'$)
10:             $Vertex$ $v'$ = new $Vertex(v.oids \cup \{op'.oid\}, \emptyset)$
11:             $Edge$ $e_{vv'} \leftarrow$ new $Edge(op'\langle op \rangle, v')$
12:             $v.edges[d] \leftarrow e_{vv'}$
13:             $Edge$ $e_{u'v'} \leftarrow$ new $Edge(op\langle op' \rangle, v')$
14:             $u'.edges[1 - d] \leftarrow e_{u'v'}$
15:             $u \leftarrow u'$
16:             $v \leftarrow v'$
17:             $op \leftarrow op\langle op' \rangle$
18:         end while
19:         $cur \leftarrow v$
20:         return $op$
21:     end procedure

22:     procedure Locate($op : Op$) : $Vertex$
23:         return $Vertex$ $v$ with $v.oids = op.ctx$
24:     end procedure

25:     procedure Add($op : Op, d : LG, u : Vertex$) : $Vertex$
26:         $Vertex$ $v \leftarrow$ new $Vertex(u.oids \cup \{op.oid\}, \emptyset)$
27:         $Edge$ $e \leftarrow$ new $Edge(op, v)$
28:         $u.edges[d] \leftarrow e$
29:         return $v$
30:     end procedure
31: end                                                 ▷ Class $StateSpace2D$
```

▶ **Definition 30** (2D State Space)**.** A set of vertices S is a 2D state space if and only if

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

   The second condition above models OTs in Jupiter.

D.2 The Jupiter Protocol

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 for operations generated by other clients. 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 for operations from other clients.

Jupiter is similar to CJupiter with two major differences:

1. In $\textsc{xForm}(op : Op, d : LG = \{LOCAL, GLOBAL\})$ of Jupiter, the operation sequence with which op transforms is determined by an extra parameter d; *and*
2. In Jupiter, the server propagates the transformed operation (instead of the original one it receives from a client) to other clients.

As with CJupiter, we also describe Jupiter in three parts. In the following, we omit the details that are in common with and have been explained in CJupiter.

D.2.1 Local Processing (Do of Algorithm D.5)

When client c_i receives an operation $o \in \mathcal{O}$ from a user, it

1. applies o locally;
2. generates $op \in Op$ for o and saves it along the local dimension at the end of its 2D state space DSS_{c_i}; *and*
3. sends op to the server.

D.2.2 Server Processing (Receive of Algorithm D.6)

When the server receives an operation $op \in Op$ from client c_i, it

1. transforms op with an operation sequence along the global dimension in the 2D state space DSS_{s_i} to obtain op' by calling $\textsc{xForm}(op, GLOBAL)$ (Section D.2.4);
2. applies op' locally;
3. for each $j \neq i$, saves op' at the end of DSS_{s_j} along the global dimension; *and*
4. sends op' (instead of op) to other clients.

Algorithm D.5 Client in Jupiter.

```
1: Class $Client$ begin
2:     Var $cid : CID$
3:     Var $seq : SEQ = 0$
4:     Var $state : \Sigma = \langle\rangle$
5:     Var $S : StateSpace2D$= new $StateSpace2D()$

6:     procedure DO($o : \mathcal{O}$) : $Val$                ▷ Local Processing
7:         $(state, val) \leftarrow$ APPLY$(state, o)$
8:         $seq \leftarrow seq + 1$
9:         $Op\ op \leftarrow$ new $Op(o, (cid, seq), S.cur.oids)$
10:        $Vertex\ v \leftarrow S.$ADD$(op, LOCAL, S.cur)$
11:        $S.cur \leftarrow v$
12:        SEND$(SID, op)$                                  ▷ send $op$ to the server
13:        return $val$
14:    end procedure

15:    procedure RECEIVE($op : Op$)                          ▷ Remote Processing
16:        $Op\ op' \leftarrow S.$xFORM$(op, LOCAL)$
17:        $state \leftarrow state \circ op'.o$
18:    end procedure
19: end                                                    ▷ Class $Client$
```

Algorithm D.6 Server in Jupiter.

```
1: Class $Server$ begin
2:     Var $SS : StateSpace2D[CID]$                          ▷ one per client
3:     Var $state : \Sigma = \langle\rangle$

4:     procedure RECEIVE($op : Op$)                          ▷ Server Processing
5:         $Op\ op' \leftarrow SS[op.oid.cid].$xFORM$(op, GLOBAL)$
6:         $state \leftarrow state \circ op'.o$
7:         for all $c \in CID \setminus \{op.oid.cid\}$ do
8:             $SS[c].$ADD$(op, GLOBAL, SS[c].cur)$
9:             SEND$(c, op')$                   ▷ send $op'$ (not $op$) to client $c$
10:        end for
11:    end procedure
12: end                                                    ▷ Class $Server$
```

D.2.3 Remote Processing (Receive of Algorithm D.5)

When client c_i receives an operation $op \in Op$ from the server, it

1. transforms op with an operation sequence along the local dimension in its 2D state space DSS_{c_i} to obtain op' by calling $\textsc{xForm}(op, \mathit{LOCAL})$ (Section D.2.4); *and*
2. applies op' locally.

D.2.4 OTs in Jupiter (xForm of Algorithm D.4)

The procedure $\textsc{xForm}(op : Op, d : LG = \{\mathit{LOCAL}, \mathit{GLOBAL}\})$ of Jupiter is similar to $\textsc{xForm}(op : Op)$ of CJupiter except that in Jupiter, the operation sequence with which op transforms is determined by an extra parameter d. Specifically, it

1. locates the vertex u whose *oids* matches the operation context $op.ctx$ of op; *and*
2. iteratively transforms op with an operation sequence along the d dimension from u to the final vertex *cur* of this 2D state space.

E Proofs for Section 4: CJupiter is Equivalent to Jupiter

E.1 Proof for Proposition 14 ($n \leftrightarrow 1$)

Proof. By mathematical induction on the operation sequence $O = \langle op_1, op_2, \cdots, op_m \rangle$ the server processes.

Base Case. $k = 1$. According to the Jupiter and CJupiter protocols, it is obviously that

$$\text{CSS}_s^1 = \text{DSS}^1_{s_{c(op_1)}}.$$

Inductive Hypothesis. Suppose that $(*)$ holds for k:

$$\text{CSS}_s^k = \bigcup_{i=1}^{i=k} \text{DSS}^i_{s_{c(op_i)}}.$$

Inductive Step. We shall prove that $(*)$ holds for $(k+1)$:

$$\text{CSS}_s^{k+1} = \bigcup_{i=1}^{i=k+1} \text{DSS}^i_{s_{c(op_i)}}.$$

By inductive hypothesis, we shall prove that

$$\text{CSS}_s^{k+1} \setminus \text{CSS}_s^k = \text{DSS}^{k+1}_{s_{c(op_{k+1})}}.$$

In other words, the OTs for op_{k+1} performed by the servers in Jupiter and CJupiter are the same. This holds due to two reasons. First, under the same schedule, the matching vertex of op_{k+1} in $\text{DSS}^k_{s_{c(op_{k+1})}}$ of Jupiter is the same with that in CSS_s^k of CJupiter, determined by its operation context (or the causally-before relation of the schedule). Second, according to Lemma 7 for CJupiter and its counterpart for Jupiter, the operation sequences with which op_{k+1} transforms are the same in both protocols. ◀

E.2 Proof for Proposition 16 ($1 \leftrightarrow 1$)

Proof. By mathematical induction on the operation sequence $O^{c_i} = \langle op_1^{c_i}, op_2^{c_i}, \ldots, op_m^{c_i} \rangle$ the client c_i processes.

Base case. $k = 1$, namely, $O^{c_i} = \langle op_1^{c_i} \rangle$. No matter whether $op_1^{c_i}$ (more specifically, $op_1^{c_i}.o$) is generated by client c_i or is an operation propagated to client c_i by the server, it obviously holds that

$$\text{DSS}^1_{c_i} = \text{CSS}^1_{c_i}.$$

Inductive Hypothesis. Suppose $O^{c_i} = \langle op_1^{c_i}, op_2^{c_i}, \ldots, op_k^{c_i} \rangle$ and $(\star)$ holds for k:

$$\text{DSS}^k_{c_i} \subseteq \text{CSS}^k_{c_i}.$$

Inductive Step. Client c_i executes the $(k+1)$-*st* operation $op_{k+1}^{c_i}$. We shall prove that $(\star)$ holds for $(k+1)$:

$$\text{DSS}^{k+1}_{c_i} \subseteq \text{CSS}^{k+1}_{c_i}.$$

We distinguish two cases between $op_{k+1}^{c_i}$ being generated by client c_i or an operation propagated to client c_i by the server.

CASE 1: *The operation $op^{c_i}_{k+1}$ is generated by client c_i.* The new 2D state space $\text{DSS}^{k+1}_{c_i}$ of Jupiter (resp. n-ordered state space $\text{CSS}^{k+1}_{c_i}$ of CJupiter) is obtained by saving $op^{c_i}_{k+1}$ at the final vertex of the previous state space $\text{DSS}^{k}_{c_i}$ (resp. $\text{CSS}^{k}_{c_i}$). Since $\text{DSS}^{k}_{c_i} \subseteq \text{CSS}^{k}_{c_i}$ (by the inductive hypothesis), we conclude that $\text{DSS}^{k+1}_{c_i} \subseteq \text{CSS}^{k+1}_{c_i}$.

CASE 2: *The operation $op^{c_i}_{k+1}$ is an operation propagated to client c_i by the server.* Due to Lemmas 7 for CJupiter and its counterpart for Jupiter, the operation sequences L with which $op^{c_i}_{k+1}$ transforms at the server in both protocols are the same. Since the communication is FIFO, when client c_i receives $op^{c_i}_{k+1}$, all the operations totally ordered by '$\prec_s$' before $op^{c_i}_{k+1}$ have already been in $\text{CSS}^{k}_{c_i}$. By Proposition 9, the OTs involved in iteratively transforming $op^{c_i}_{k+1}$ with L at the server in both protocols are also performed at client c_i in CJupiter. By contrast, in Jupiter, the resulting transformed operation, denoted $op^{c_i}_{k+1}\langle L \rangle$, is propagated to client c_i, where the set of OTs performed is a subset of those involved in transforming $op^{c_i}_{k+1}$ with L. Given the inductive hypothesis $\text{DSS}^{k}_{c_i} \subseteq \text{CSS}^{k}_{c_i}$, we conclude that $\text{DSS}^{k+1}_{c_i} \subseteq \text{CSS}^{k+1}_{c_i}$. ◀

E.3 Proof for Theorem 17 (Equivalence of Clients)

Proof. Note that in the proof for Proposition 16, no matter whether the operation $op^{c_i}_{k}$ is generated by client c_i or is an operation propagated to client c_i by the server, the final transformed operations executed at c_i in Jupiter and CJupiter are the same. ◀