Jupiter Made Abstract, and then Refined

Hengfeng Wei, Yu Huang, Ruize Tang, and Jian Lu

State Key Laboratory for Novel Software Technology, Nanjing University, China
{hfwei, yuhuang, lj}@nju.edu.cn tangruize97@gmail.com

Abstract. In the literature, there is a family of OT-based (Operational Transformation) *Jupiter* protocols for replicated lists, including *AJupiter*, *XJupiter*, and *CJupiter*. They are hard to understand due to the subtle OT technique, and little work has been done on formal verification of complete *Jupiter* protocols. Worse still, they use quite different data structures. It is unclear how they are related to each other, and it would be laborious to verify each *Jupiter* protocol separately. In this work, we make contributions towards a better understanding of *Jupiter* protocols and the relation among them. We first identify the key OT issue in *Jupiter* and present a generic solution. We summarize several techniques for carrying out the solution and propose an implementation-independent *AbsJupiter* protocol. Then, we establish the (data) refinement relation among these *Jupiter* protocols (*AbsJupiter* included). We also formally specify and verify the family of *Jupiter* protocols and the refinement relation among them using TLA$^+$ and TLC.

Keywords: Operational Transformation · Jupiter · Refinement · TLA$^+$

1 Introduction

Collaborative text editing systems, such as Google Docs [2], Firepad [1], Overleaf [5], and SubEthaEdit [6], allow 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 and updates are propagated asynchronously [10] [8].

The *replicated list object* is frequently used to model the core functionality (e.g., insertion and deletion) of replicated collaborative text editing systems [10] [21] [30] [8]. A common specification for it is the strong eventual consistency [24]. It requires that whenever two replicas have processed the same set of updates, they have the same list. A family of *Jupiter* protocols for implementing such a replicated list have been proposed, including *AJupiter* [9], *XJupiter* [30], and *CJupiter* [29]. They adopt the client/server architecture, where the server serializes operations and propagates them from one client to others (Figure 1) [1]. To achieve convergence, *Jupiter* adopts the OT (Operational Transformation) technique [10] [26] to resolve the conflicts caused by concurrent operations. The

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

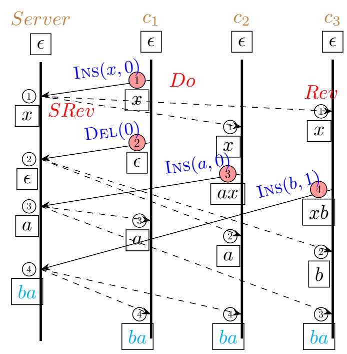

Fig. 1: System model. The circled numbers indicate the *serialization order* (SO) in which the operations are received at the server. (Adapted from [29].)

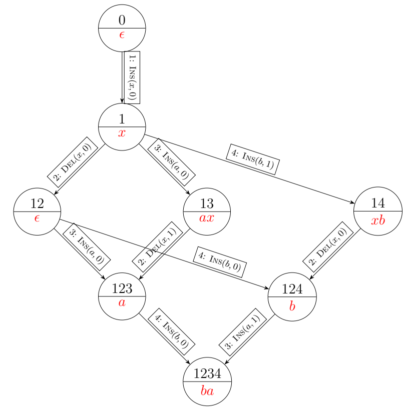

Fig. 2: The same n-ary digraph constructed by *CJupiter* for each replia under the schedule of events in Figure 1. (Adapted from [29].)

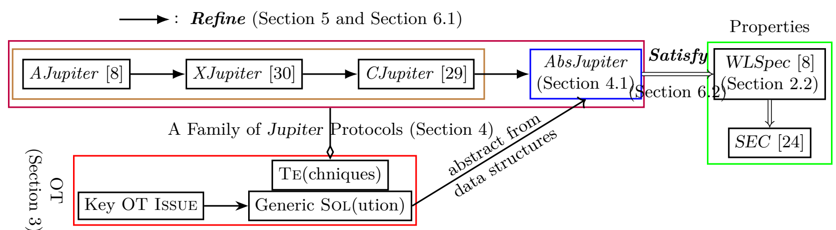

Fig. 3: The overview of contributions.

idea of OT is, for each replica, to process local operations immediately and to *transform* received operations according to the effects of previously processed concurrent operations. The transformation rules are called OT functions [10] [21]. Consider, for example, a replicated list system with two client replicas C_1 and C_2 which initially hold the same list "ab". Suppose that user 1 issues $o_1 = \text{INS}(1, x)$ at C_1 and concurrently user 2 issues $o_2 = \text{DEL}(2)$ at C_2 [2]. After being executed locally, each operation is sent to the other replica. Without OT, C_1 and C_2 wind up with different lists (i.e., "xb" and "xa", respectively). With OT, o_2 is transformed to $o_2' = \text{DEL}(3)$ at C_1, taking into account the fact that o_1 has inserted an element at position 1. Meanwhile, o_1 remains unchanged after OT at C_2. As a result, two replicas converge to the same list "xa".

When several replicas diverge by multiple operations, OT becomes much more subtle and error-prone. Some published OT-based protocols are even later shown incorrect [23] [13]. The intrinsic complexity in concurrency control makes

[2] The positions are indexed from 1.

the OT-based *Jupiter* protocols hard to understand. Moreover, little has been done on formal verification of complete OT-based protocols (not only of OT functions). Worse still, *Jupiter* protocols use quite different data structures, rendering the relation among them unclear. It would be also laborious and wasteful to prove or verify that the *Jupiter* protocols satisfy a certain property one by one. In this work, we make contributions towards a better understanding of *Jupiter* protocols and the relation among them. Specifically (Figure 3),

- (Section 3) We first identify the key issue involving OT *Jupiter* needs to address as follows: When a replica r receives an operation op, which operations should op be transformed against and in what order before it is applied? We also present a generic solution to this issue: Transform op against the set of concurrent operations previously executed at r in the serialization order established at the server. Then, we summarize several techniques the *Jupiter* protocols adopt to carry out the solution, including (TECH-I) those for deciding whether two operations are concurrent, (TECH-II) for determining the serialization order, and (TECH-III) the data structures to maintain (intermediate) OT results and to guide OTs.
- (Section 4.1) We propose *AbsJupiter*, an abstract *Jupiter* protocol which captures the OT essence of existing *Jupiter* protocols. Specifically, it addresses the key OT issue in a way abstract from concrete data structures regarding TECH-III by using mathematical sets.
- (Section 5) For different purposes such as performance or ease of correctness proof, existing *Jupiter* protocols use quite different data structures. The implementation details in TECH-III have obscured the similarities among them in TECH-I and TECH-II. We show that the existing *Jupiter* protocols are actually (data) refinements [11] [18] [17] of *AbsJupiter* in data structures. Specifically, we show that *AJupiter* is a refinement (a.k.a. implementation) of *XJupiter*, which is a refinement of *CJupiter*, which is a refinement of *AbsJupiter*. As a consequence, the properties hold for *AbsJupiter* also automatically hold for other *Jupiter* protocols.
- (Sections 4 and 6) In Section 4, we formally specify the family of *Jupiter* protocols in TLA$^+$ [16]. The refinement mappings among *Jupiter* protocols are also expressed in TLA$^+$ in Section 5. Section 6 presents the model checking results conducted by TLC [31] of verifying both properties for *Jupiter* protocols and the refinement relations among them.

Section 2 covers preliminaries on system model, OT, and list specifications. Section 7 discusses related work. Section 8 concludes. Appendix A provides a brief introduction to TLA$^+$. Appendix B contains more TLA$^+$ code. (The whole project can be found in [3].) Appendix C shows more model checking results.

2 Preliminaries

2.1 System Model

We let *Client* denote the set of n client replicas, *Server* the unique server replica, and *Replica* the set of all replicas (Figure 1). Client replicas are connected to the server replica via FIFO channels. The set of messages is denoted by M. A replica is modelled as a state machine. Each replica r maintains its current list $list[r]$ and interacts with three kinds of events from users and other replicas:

- $Do(c \in Client, op \in Op)$: Client c receives an operation $op \in Op$ (defined below) from an unspecified user [3] and responds to the user immediately. It then sends the update in an message $m \in M$ to the server asynchronously.
- $Rev(c \in Client, m \in M)$: Client c receives a message m from the server.
- $SRev(m \in M)$: The server receives a message m from a client.

2.2 List, OT Functions, and Weak List Specification (*WLSpec*)

A replicated list object supports two types of update operations (in Op): Del and Ins, defined as records as follows. Following [8], we assume that all inserted elements are unique, which can be achieved by attaching replica identifiers and local sequence numbers. The priority field "*pr*" of Ins helps to resolve the conflicts caused by two concurrent Ins operations that are intended to insert different elements at the same position.

MODULE OT

$Del \triangleq [type : \{\text{"Del"}\}, pos : Nat]$ The positions (pos) are indexed from 1.
$Ins \triangleq [type : \{\text{"Ins"}\}, pos : Nat, ch : Char, pr : 1 \,..\, Cardinality(Client)]$
$Op \triangleq Ins \cup Del$ The set of all possible update operations.

$OTID(ins, del) \triangleq$ ins is transformed against del; I for Ins and D for Del.
$\quad \text{IF } ins.pos \leq del.pos \text{ THEN } ins \text{ ELSE } [ins \text{ EXCEPT } !.pos = @ - 1]$
$OT(lop, rop) \triangleq$ Calls $OTII$, $OTID$, $OTDI$, or $OTDD$ to transform lop against rop.

$OT(lop, rop)$ transforms lop against rop by calling the appropriate OT function according to the types of lop and rop. For example, $OTID$ defines how an Ins operation ins is transformed against a Del operation del. It adjusts the insertion position of $lins$ according to the deletion position of del. A complete definition of OT functions for lists can be found in Appendix B [10] [21].

We consider the *weak list specification* $WLSpec$ [8], which is stronger than the strong eventual consistency (SEC) [24]. $WLSpec$ is equivalent to the *pairwise state compatibility property* [29]. It requires any pair of lists across the system to be compatible, where two lists l_1 and l_2 are compatible if and only if for any two common elements e_1 and e_2 of l_1 and l_2, their relative orderings are the same in l_1 and l_2; see Appendix B for its TLA$^+$ description.

3 *Jupiter* Family

The key issue (Issue) for *Jupiter* protocols to address is as follows: When a replica r receives an operation op, which operations should op be transformed against and in what order before it is applied? The solution (Sol) is to transform

[3] We also sometimes say that client c generates the operation op.

Table 1: Techniques adopted by *Jupiter* protocols to carry out SOL.

Protocols	(TECH-I) *Concurrent Ops.*	(TECH-II) *SO Order*	(TECH-III) *Data Structures*
AbsJupiter	COT	SV	Set
CJupiter	COT	SV	n-ary digraph
XJupiter	COT	COT	$2D$ digraph
AJupiter	ACK	Buffer	$1D$ buffer

op against the set of operations that are *concurrent* with it and have been previously executed at r in their *serialization order*, denoted SO, i.e., the order they are received by the server. The four *Jupiter* protocols we study differ in the way they carry out the solution. Table 1 summarizes several key techniques they adopt to carry out the solution, including (TECH-I) those for deciding whether two operations are concurrent, (TECH-II) for determining the serialization order, and (TECH-III) the data structures to maintain OT results and to guide OTs.

3.1 Context-based OT (COT)

According to whether they use context-based operations (Cop) and context-based OT (COT) [28], *Jupiter* protocols fall into two categories: *AbsJupiter*, *CJupiter*, and *XJupiter* are all context-based, while *AJupiter* is not.

Each operation $op \in Op$ is associated with a unique operation identifier (oid, for short) in Oid, a record of client c that generates op and a local sequence number $cseq[c]$ of c. Each replica r maintains its *document state* $ds[r]$ (initially $\{\}$) as the set of operation identifiers it has processed. The document state $ds[r]$ is updated to include oid whenever the replica r receives an operation with oid.

MODULE COT

$Oid \triangleq [c : Client,\ seq : Nat]$ — the client that generates cop

$Cop \triangleq [op : Op,\ oid : Oid,\ ctx : \text{SUBSET}\ Oid]$ — $ClientOf(cop) \triangleq cop.oid.c$

$COT(lcop, rcop) \triangleq$ Precondition: $lcop.ctx = rcop.ctx$

$\quad [lcop\ \text{EXCEPT}\ !.op = OT(lcop.op, rcop.op),\ !.ctx = @ \cup \{rcop.oid\}]$

A *context-based operation* $cop \in Cop$ is a record of operation $op \in Op$, its oid $oid \in Oid$, and its context $ctx \subseteq Oid$ representing a document state. When an operation is generated by client c, its context is set to be the current document state $ds[c]$ of c. When a context-based operation $lcop$ is transformed against another one $rcop$, $lcop.ctx$ will be updated to include $rcop.oid$. Note that according to the context-based condition (CC) [28], two context-based operations can be transformed against each other, only if they have the same context.

3.2 Serial Views (SV)

In *AbsJupiter* and *CJupiter*, replicas need to decide the SO order among operations with local knowledge. To do this, each replica r maintains a *serial* *view* $serial[r] \in Seq(Oid)$ about SO. The server always has the latest serial view $serial[Server]$ and updates it in *SRev* by each time appending to it the received operation identifier. Each client c synchronizes its serial view $serial[c]$ with the server in $Rev(c)$.

The operator $so(oid1, oid2, sv)$ in SV (Appendix B) decides whether $oid1$ precedes (or will precede) $oid2$ in SO order given the serial view sv of some replica. There are three cases: CASE (1) If both have been at the server (i.e., both $oid1$ and $oid2$ are in sv), we use the order they arrive the server, which is captured by sv; CASE (2) If none have been at the server, they must be generated by the same client and we use the order they were generated; CASE (3) Otherwise, the one has been at the server precedes the other has not.

3.3 Data Structures

Set In *AbsJupiter*, each replica r maintains a *set* $copss[r]$ (initially $\{\}$) of context-based operations (TECH-III). When a replica r receives a context-based operation *cop*, it calls $xForm(r, cop)$ to transform *cop* against a subset of context-based operations in $copss[r]$ that are *concurrent* with *cop* in their SO order (SOL).

```
------------------------------ MODULE Set ------------------------------
xForm(r, cop) ≜   Transform cop at replica r
  LET ctxDiff ≜ ds[r] \ cop.ctx   calculate concurrent operations
      xFormHelper(coph, ctxDiffh, copssh) ≜   Return transformed xcop and
        IF ctxDiffh = {} THEN [xcop ↦ coph, xcopss ↦ copssh]   new copss
        ELSE LET foidh ≜ CHOOSE oid ∈ ctxDiffh :   the first oid in ctxDiffh
                           ∀ id ∈ ctxDiffh \ {oid} : so(oid, id, serial[r])
                 fcoph ≜ CHOOSE fcop ∈ copss[r] :
                           fcop.oid = foidh ∧ fcop.ctx = coph.ctx   CC
                 xcoph ≜ COT(coph, fcoph)    xfcoph ≜ COT(fcoph, coph)
             IN  xFormHelper(xcoph, ctxDiffh \ {fcoph.oid},
                                    copssh ∪ {xcoph, xfcoph})
  IN  xFormHelper(cop, ctxDiff, copss[r] ∪ {cop})
-------------------------------------------------------------------------
```

Due to the FIFO communication, we have that $cop.ctx \subseteq ds[r]$. Thus, *xForm* first calculates the set of (oids of) concurrent operations with *cop* as the set difference *ctxDiff* between $ds[r]$ and $cop.ctx$ (TECH-I). Then it recursively transforms *cop* against the context-based operations in $copss[r]$ whose oids are in *ctxDiff* in their SO order according to the serial view $serial[r]$. This is done in $xFormHelper(coph \leftarrow cop, ctxDiffh \leftarrow ctxDiff, copssh \leftarrow copss[r] \cup \{cop\})$:

1. If *ctxDiffh* is empty, the most recently transformed *coph* and the latest data structure *copssh* are returned.
2. Otherwise, *xFormHelper* chooses the next operation *fcoph* against which *coph* is to be transformed, such that $fcoph.oid$ is the *first* one in the current *ctxDiffh* (TECH-II) and that $fcoph.ctx = coph.ctx$ (the CC condition).
3. *coph* and *fcoph* are transformed against each other. The intermediate transformed operation *xcoph* is recursively transformed against the remaining concurrent operations (with *oid*) in $ctxDiffh \setminus \{foph.oid\}$.

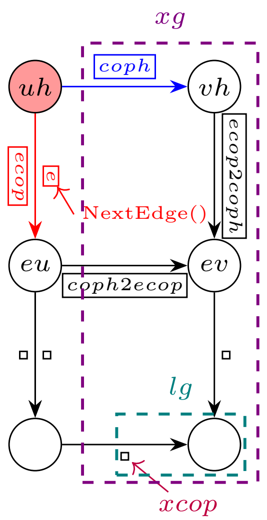

Fig. 4: *xForm* of *Digraph*.

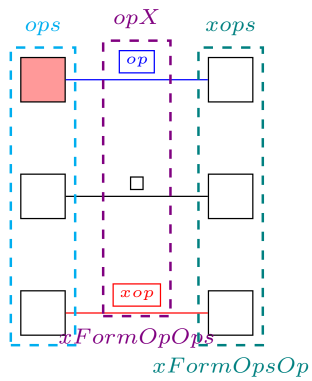

Fig. 5: *xForm* of *Buffer*.

Digraph In *CJupiter* and *XJupiter*, the set of context-based operations are organized into edge-labeled *digraphs* (Tech-III). A digraph is represented by a record with *node* field and *edge* field. Each node in $G.node$ of a digraph G represents a document state. Each directed edge e in $G.edge$ is labeled with a context-based operation *cop* satisfying $cop.ctx = e.from$, meaning that when applied, *cop* changes the document state from $e.from$ to $e.to = e.from \cup \{cop.oid\}$; see Figure 2 for an illustration. The operator $\oplus$ takes union of two digraphs.

MODULE *Digraph*

$IsGigraph(G) \triangleq$ G is a record with *node* field and *edge* field

$\quad \land\ G.node \subseteq (\text{SUBSET } Oid)$ each node represents a document state

$\quad \land\ G.edge \subseteq [from : G.node,\ to : G.node,\ cop : Cop]$ label: *cop*

$EmptyGraph \triangleq [node \mapsto \{\{\}\},\ edge \mapsto \{\}]$

$g \oplus h \triangleq [node \mapsto g.node \cup h.node,\ edge \mapsto g.edge \cup h.edge]$ union

In *CJupiter* and *XJupiter*, when a replica r receives a context-based operation *cop*, it calls $xForm(NextEdge, r, cop, g)$ to iteratively transform *cop* against a sequence of context-based operations along a *path* in some digraph g maintained by r (Sol). This path starts with the node u equal to $cop.ctx$ and ends with the one equal to $ds[r]$. Each such path contains the operations whose oids are in $ds[r] \setminus cop.ctx$, which are concurrent with *cop* due to the FIFO communication (Tech-I). The next edge is chosen by *NextEdge* specific to *CJupiter* and *XJupiter* to ensure the so order (Tech-II). $xFormHelper(uh, vh, coph, gh)$ starts the transformation with $uh \leftarrow u$ (Figure 4; see Appendix B for code):

1. If $uh = ds[r]$, the most recently transformed operation $coph$, the extra digraph gh produced in $xForm$ so far, and the node and edge (combined in lg) produced in the last iteration of transformation are returned.
2. Otherwise, the next edge e (and its associated operation $ecop \triangleq e.cop$) outgoing from uh is chosen using $NextEdge(r, uh, g)$ specific to *CJupiter* and *XJupiter* (TECH-II).
3. $coph$ and $ecop$ are transformed against each other. The intermediate transformed operation $coph2ecop$ is then recursively transformed against the sequence of operations starting with the node $eu \triangleq e.to$, the successor of uh along the edge e.

Buffer *AJupiter* maintains *buffers* (i.e., sequences) of operations of type Op (TECH-III). $xForm(op, ops)$ transforms an operation op against a buffer ops of operations. It utilizes $xFormOpOps\ (op, ops)$ and $xFormOpsOp(ops, op)$ to obtain the last transformed operation xop and the transformed buffer $xops$, respectively; see Figure 5. $xFormShift(op, ops, shift)$ transforms op against the subsequence of ops obtained by shifting the first $shift$ operations out of ops.

```
──────────────────────── MODULE Buffer ────────────────────────
xFormOpOps(op, ops) ≜   Transform op against ops
    IF ops = ⟨⟩ THEN ⟨op⟩  and return intermediate transformed operations.
     ELSE ⟨op⟩ ∘ xFormOpOps(OT(op, Head(ops)), Tail(ops))
xFormOpsOp(ops, op) ≜   Transform ops against op and return the transformed ops.
    LET opX ≜ xFormOpOps(op, ops)
    IN  [i ∈ 1 .. Len(ops) ↦ OT(ops[i], opX[i])]
xForm(op, ops) ≜  see Figure 5
    [xop ↦ Last(xFormOpOps(op, ops)), xops ↦ xFormOpsOp(ops, op)]
xFormShift(op, ops, shift) ≜ xForm(op, SubSeq(ops, shift + 1, Len(ops)))
───────────────────────────────────────────────────────────────
```

4 *Jupiter* Protocols

In this section, we formally specify *Jupiter* protocols in TLA$^+$. We focus on when and how OTs are performed and the data structures (TECH-III) supporting OTs. The techniques of COT and SV are orthogonal to this and are omitted here.

4.1 *AbsJupiter*

In *AbsJupiter*, each replica r maintains a set $copss[r]$ (initially $\{\}$) of context-based operations. The operator $Perform(r, cop)$ calls $xForm(r, cop)$ to transform cop in $copss[r]$. The transformed operation $xform.xcop.op$ is applied to $list[r]$ and $copss[r]$ is updated to $xform.xcopss$.

In $Do(c, op)$, the client c first wraps op into a context-based operation cop by attaching oid and $ctx = ds[c]$ to it. Then it updates $copss[c]$ to include cop, applies op to $list[c]$, and sends cop to the $Server$. When the server receives a context-based operation cop from client c, it calls $Perform(Server, cop)$ and then broadcasts *cop* to other clients than c; see $SRev(cop)$. In $Rev(c, cop)$, the client c just calls $Perform(c, cop)$.

MODULE *AbsJupiter*

VARIABLES *copss* — $copss[r]$: the set of context-based operations maintained at replia r

$$
\begin{aligned}
&Perform(r,\ cop) \triangleq \text{LET } xform \triangleq xForm(r,\ cop) \quad \boxed{xform : [xcop, xcopss]}\\
&\qquad \text{IN } \land\ copss' = [copss \text{ EXCEPT } ![r] = xform.xcopss]\\
&\qquad\qquad \land\ \boxed{\text{apply } xform.xcop.op \text{ to } list[r]}\\
&Do(c,\ op) \triangleq \text{LET } cop \triangleq [op \mapsto op,\ oid \mapsto [c \mapsto c,\ seq \mapsto cseq[c]],\ ctx \mapsto ds[c]]\\
&\qquad \text{IN } \land\ copss = [copss \text{ EXCEPT } ![c] = @ \cup \{cop\}]\\
&\qquad\qquad \land\ \boxed{\text{apply } op \text{ to } list[c]\text{; send } cop \text{ to the } Server}\\
&Rev(c,\ cop) \triangleq Perform(c,\ cop)\\
&SRev(cop) \triangleq \land\ Perform(Server,\ cop)\\
&\qquad\qquad \land\ \boxed{\text{broadcast } cop \text{ to clients other than } ClientOf(cop)}
\end{aligned}
$$

4.2 *CJupiter*

In *CJupiter*, each replica r maintains an n-ary digraph $css[r]$ (initially *EmptyGraph*), a digraph where the outdegree of each node can be at most n. (See Figure 2 for illustration and Appendix B for code.) In $Do(c, op)$, the client c first wraps *op* into a context-based operation *cop*. Then it applies *op* to $list[c]$, appends *cop* to $ds[c]$ in $css[c]$, and sends *cop* to the server. The definitions of *Rev* and *SRev* of *CJupiter* are the same with those in *AbsJupiter*, except that $xForm(NextEdge, r, cop, g \leftarrow css[r])$ is called by replica r to transform *cop* against a sequence of context-based operations with *cop* along a *path* in digraph $css[r]$. The next edge from a given node chosen in *NextEdge* is the *first* one in terms of SO according to the serial view $serial[r]$ of r. The intermediate digraph $xform.xg$ produced in *xForm* is integrated into $css[r]$ and the transformed operation $xform.xcop.op$ is applied to $list[r]$.

4.3 *XJupiter*

XJupiter uses $2D$ digraphs where the outdegree of each node is at most 2. Each client c maintains a single $2D$ digraph $c2ss[c]$, and the server maintains n $2D$ digraphs, one digraph $s2ss[c]$ per client c. Conceptually, a $2D$ digraph, either $c2ss[c]$ or $s2ss[c]$, has two dimensions: a local dimension for storing operations generated by c and a global dimension by others.

In $Do(c, op)$, the client c first wraps *op* into a context-based operation *cop* by attaching *oid* and $ctx = ds[c]$ to it. Then it applies *op* to $list[c]$, appends *cop* to $ds[c]$ along the *local* dimension of $c2ss[c]$, and sends *cop* to the server.

When the server receives a context-based operation *cop* from client c, it transforms *cop* against the context-based operations along the *remote* dimension from node $u \triangleq cop.ctx$ to $ds[Server]$ in $s2ss[c]$. In $SRev(cop)$, this is done in $xForm(NextEdge, Server, cop, s2ss[c])$, where *NextEdge* returns the *unique* outgoing edge of a given node. Then, the transformed operation *xform.xcop.op* is applied to *list*[*Server*], *s2ss*[*c*] is updated to integrate *xform.xg*, and *xform.lg* is appended to the *remote* dimension of each digraph $s2ss[cl \neq c]$. Finally, the server broadcasts the *transformed* operation *xform.xcop* to other clients than c.

In $Rev(c, cop)$, the client c calls $xForm(NextEdge, c, cop, c2ss[c])$ to transform *cop* against the operations along the *local* dimension from node $u \triangleq cop.ctx$ to $ds[c]$ in $c2ss[c]$. The intermediate digraph *xform.xg* produced is integrated into $c2ss[c]$ and the transformed operation *xform.xcop.op* is applied to $list[c]$.

```
──────────────────────── MODULE XJupiter ────────────────────────
VARIABLES c2ss,   c2ss[c]: the 2D digraph maintained at client c
          s2ss    s2ss[c]: the 2D digraph maintained by the Server for client c
─────────────────────────────────────────────────────────────────
NextEdge(_, u, g) ≜ CHOOSE e ∈ g.edge : e.from = u
Do(c, op) ≜ LET cop ≜ [op ↦ op, oid ↦ [c ↦ c, seq ↦ cseq[c]], ctx ↦ ds[c]]
                u ≜ ds[c]    v ≜ u ∪ {cop.oid}
            IN  ∧ c2ss' = [c2ss EXCEPT ![c] =   append cop to u ≜ ds[c]
                    @ ⊕ [node ↦ {v}, edge ↦ {[from ↦ u, to ↦ v, cop ↦ cop]}]]
                ∧ apply op to list[c]; send cop to the Server
Rev(c, cop) ≜ LET xform ≜ xForm(NextEdge, c, cop, c2ss[c])  xform: [xcop, xg, lg]
              IN  ∧ c2ss' = [c2ss EXCEPT ![c] = @ ⊕ xform.xg]
                  ∧ apply xform.xcop.op to list[c]
SRev(cop) ≜
    LET c ≜ ClientOf(cop)
    xform ≜ xForm(NextEdge, Server, cop, s2ss[c])  xform: [xcop, xg, lg]
    IN    ∧ s2ss' = [cl ∈ Client ↦ IF cl = c THEN s2ss[cl] ⊕ xform.xg
                                             ELSE s2ss[cl] ⊕ xform.lg]
          ∧ apply xform.xcop.op to list[Server]
          ∧ broadcast the transformed operation xform.xcop to clients other than c
─────────────────────────────────────────────────────────────────
```

4.4 *AJupiter*

In *AJupiter*, each client c maintains a buffer $cbuf[c]$ for storing the operations (maybe transformed) it generates, and a counter $crec[c]$ counting the number of operations it has received from the server since the last time it generated an operation and sent a message. Similarly, the server maintains for each client c a buffer $sbuf[c]$ for storing the (transformed) operations generated by other clients than c, and a counter $srec[c]$ counting the number of operations the server has received from client c since the last time an operation which is generated by other clients than c was transformed at the server and a message was broadcast.

The counters (i.e., $crec[c]$ and $srec[c]$) are piggybacked in the *ack* field in messages *AJMsg* telling the other side how many new messages have been received since the last time a message was sent [9]; see Appendix B for code. When a client c receives a message m of form $[ack \mapsto srec[c], op]$ broadcast by the *Server*, it knows that *op* is generated by another client and more importantly that the set of operations against which *op* has been transformed at the

Server contains the first *ack* operations in $cbuf[c]$. Thus, in $Rev(c, m)$, client c calls $xFormShift(m.op, cbuf[c], m.ack)$ to transform *op* against the subsequence of operations obtained by shifting the first $m.ack$ operations out of $cbuf[c]$. Similarly, when the *Server* receives a message m of form $[c, ack \mapsto crec[c], op]$ from client c, it knows that among the (transformed) operations in $sbuf[c]$ generated by other clients than c, the first *ack* operations have been broadcast to c and have been transformed at c before *op* was generated. Thus, in $SRev(m)$, the *Server* calls $xFormShift(m.op, sbuf[c], m.ack)$ to transform *op* against the subsequence of operations obtained by shifting the first $m.ack$ operations out of $sbuf[c]$. The transformed operation *xop* will be appended to other $sbuf[cl]$ for clients $cl \neq c$. Finally, the *Server* sends the *transformed* operation *xop* along with $srec[cl]$ to client $cl \neq c$.

5 Refinement

The OT behaviors (namely, when and how to perform OTs) of four *Jupiter* protocols are essentially the same under the same schedule of events of *Do*, *Rev*, and *SRev*. The main difference lies in the data structures they use to support OTs. *AbsJupiter* maintains *sets* of context-based operations. *CJupiter* organizes these operations into *n-ary digraphs*. *XJupiter* synchronizes each client with its counterpart at the server, where *2D digraphs* that distinguish local dimension from remote dimension are sufficient. *AJupiter* separately maintains the local dimension and the remote dimension at clients and their counterparts at the server, respectively, thus reducing $2D$ digraphs to *1D buffers*. In this section, we establish the (data) refinement relation [11] [18] [17] among these *Jupiter* protocols. Specifically, we show that *AJupiter* is a refinement of *XJupiter*, which is a refinement of *CJupiter*, which is a refinement of *AbsJupiter*, by defining (data) refinement mappings to simulate the data structure of one *Jupiter* protocol using that of another one. In the following, we focus on refinement mappings for data structures mentioned above, and omit details for other variables.

5.1 *CJupiter* Refines *AbsJupiter*

The set $copss[r]$ of context-based operations maintained at replica r in *AbsJupiter* has been organized into an n-ary digraph $css[r]$ in *CJupiter*, by matching their contexts. Therefore, the refinement mapping from *CJupiter* to *AbsJupiter* only needs to simulate $copss[r]$ in *AbsJupiter* by extracting the context-based operations associated with the edges of $css[r]$ in *CJupiter*.

MODULE *CJupiterImplAbsJupiter*

EXTENDS *CJupiter*

$AbsJ \triangleq$ INSTANCE *AbsJupiter*

  WITH $copss \leftarrow [r \in Replica \mapsto \{e.cop : e \in css[r].edge\}]$

5.2 *XJupiter* Refines *CJupiter*

The refinement mapping defined in *XJupiterImplCJupiter* simulates, for each replica, the n-ary digraph in *CJupiter* using the $2D$ digraph(s) in *XJupiter*.

At the server side, *XJupiter* has decomposed the single n-ary digraph $css[Server]$ in *CJupiter* into n $2D$ digraphs, one $s2ss[c]$ for each client c. Thus, the refinement mapping simulates $css[Server]$ by taking *union* of these $s2ss[c \in Client]$. This is expressed in TLA$^+$ as $css[Server] \leftarrow SetReduce(\oplus, Range(s2ss), EmptyGraph)$, where $Range(s2ss)$ is the set of $s2ss[c]$ for all c, and *SetReduce* combines $Range(s2ss)$ into one using $\oplus$ with an empty digraph as initial value.

The server in *XJupiter* broadcasts the *transformed* operation $xform.xcop$ (instead of cop it receives) to clients. Thus, the clients can skip the OTs transforming cop to $xform.xcop$ performed at the server. To simulate the n-ary digraph $css[c]$ at client c in *CJupiter* using the $2D$ digraph $c2ss[c]$ in *XJupiter*, we need to *complement* $c2ss[c]$ with those OTs skipped by *XJupiter*. To this end, we introduce two auxiliary variables in *XJupiterImplCJupiter* to record OTs. The variable $op2ss$ is a function mapping an operation (identifier) to the extra $2D$ digraph produced during it is transformed at the server. When an operation cop is transformed at the server, the new mapping $cop.oid :> xform.xg$ is added to $op2ss$; see $SRevImpl(cop)$. When a client c receives the transformed operation $xform.xcop$ broadcast by the server, it accumulates this extra $2D$ digraph $op2ss[cop.oid]$ into $c2ssX[c]$, the overall $2D$ digraph that has been skipped by client c; see $RevImpl(c, cop)$. Thus, for client c, the simulation between $css[c]$ and $c2ss[c]$ can be expressed as $css[c] \leftarrow c2ss[c] \oplus c2ssX[c]$.

MODULE *XJupiterImplCJupiter*

EXTENDS *XJupiter*
VARIABLES $op2ss$, a function mapping an operation (identifier) to the $2D$ digraph produced during it is transformed at the server
$c2ssX$ $c2ssX[c]$: $2D$ digraph that has been skipped by client c

$$RevImpl(c, cop) \triangleq c2ssX' = [c2ssX \text{ EXCEPT } ![c] = @ \oplus op2ss[cop.oid]]$$
$$SRevImpl(cop) \triangleq \text{LET } xform \triangleq xForm(NextEdge, Server, cop, s2ss[ClientOf(cop)]) \text{ IN } op2ss' = op2ss \mathbin{@@} cop.oid :> xform.xg$$

$$CJ \triangleq \text{INSTANCE } CJupiter \text{ WITH } ss \leftarrow [r \in Replica \mapsto \text{IF } r = Server \text{ THEN } SetReduce(\oplus, Range(s2ss), EmptySS) \text{ ELSE } c2ss[r] \oplus c2ssX[r]]$$

5.3 *AJupiter* Refines *XJupiter*

AJupiter uses $1D$ buffers to replace $2D$ digraphs in *XJupiter*, by keeping only the latest operation sequences that should participate in further OTs and discarding the old ones and intermediate transformed operations. Therefore, the refinement mapping needs to reconstruct these $2D$ digraphs in *XJupiter* from the OTs performed on $1D$ buffers in *AJupiter*. To this end, we introduce two auxiliary variables $c2ss$ and $s2ss$ in $AJupiterImplXJupiter$ which are to simulate $c2ss$ and $s2ss$ in $XJupiter$, respectively; see the definition of XJ. These two auxiliary variables are supposed to be updated in accordance with $cbuf$ and $sbuf$ of $AJupiter$. Specifically, in $DoImpl(c, op)$, the generated operation op is wrapped as a context-based operation cop and added to $c2ss[c]$ as in $XJupiter$, besides it is stored in $cbuf[c]$ as in $AJupiter$ (not shown here). In $RevImpl(c, m)$ and $SRev(m)$, $xFormCopCopsShift$ behaves as $xFormShift$ and $xFormOpOps$ used in $AJupiter$, except that the former performs COTs on context-based operations and stores intermediate digraph produced during COTs into $c2ss[c]$ and $s2ss$ as in $XJupiter$, respectively.

$$
\begin{array}{l}
\textsc{module}\ AJupiterImplXJupiter \\
\textsc{extends}\ AJupiter \\
\textsc{variables}\ c2ss,\ s2ss \\
\hline
DoImpl(c,\ op) \triangleq \textsc{let}\ cop \triangleq [op \mapsto op,\ oid \mapsto [c \mapsto c,\ seq \mapsto cseq[c]],\ ctx \mapsto ds[c]] \\
\qquad \textsc{in}\ \ c2ss' = [c2ss\ \textsc{except}\ ![c] = \\
\qquad\qquad @ \oplus [node \mapsto \{ds'[c]\}, \\
\qquad\qquad\quad edge \mapsto \{[from \mapsto ds[c],\ to \mapsto ds'[c],\ cop \mapsto cop]\}]] \\
RevImpl(c,\ m) \triangleq \textsc{let}\ xform \triangleq xFormCopCopsShift(m.cop,\ cbuf[c],\ m.ack) \\
\qquad \textsc{in}\ \ c2ss' = [c2ss\ \textsc{except}\ ![c] = @ \oplus xform.xg] \\
SRevImpl(m) \triangleq \textsc{let}\ c \triangleq ClientOf(m.cop) \\
\qquad\qquad xform \triangleq xFormCopCopsShift(m.cop,\ sbuf[c],\ m.ack) \\
\qquad \textsc{in}\ \ s2ss' = [cl \in Client \mapsto \textsc{if}\ cl = c\ \textsc{then}\ s2ss[cl] \oplus xform.xg \\
\qquad\qquad\qquad\qquad\qquad\qquad \textsc{else}\ \ s2ss[cl] \oplus xform.lg] \\
\hline
XJ \triangleq \textsc{instance}\ XJupiter\ \textsc{with}\ c2ss \leftarrow c2ss,\ s2ss \leftarrow s2ss
\end{array}
$$

6 Model Checking Results

In this section, we first present the model checking results of verifying the refinement relation among *Jupiter* protocols defined in Section 5. Thanks to the refinement relation, we then only need to verify $AbsJupiter$ with respect to desired properties to ensure the correctness of all *Jupiter* protocols.

The model checking is conducted by TLC [31] (of version 1.5.7), a model checker for TLA$^+$, on a 2.40GHz 6-core machine with 64GB RAM. For each group of model checking experiments, we vary the number of clients and the number of characters allowed to insert [4]. We use symmetry set [16] for the set *Char* of characters. The initial lists on all replicas are empty. We use 10 threads and report the following statistics: the diameter of the reachable-state graph

[4] The positive model checking results help to gain great confidence in the correctness of these *Jupiter* protocols and the refinement relation among them, given the empirical study [32] that "almost all failures (of 198 production failures in distributed data-intensive systems) require only 3 or fewer nodes to reproduce".

Table 2: Model checking results of verifying that *CJupiter* refines *AbsJupiter*.

TLC Model ($\#Clients, \#Chars$)	**Diameter**	**# States**	**# Distinct States**	**Checking Time** ($hh : mm : ss$)
$(2, 2)$	19	50215	28307	$0 : 00 : 05$
$(2, 3)$	28	150627005	75726121	$4 : 37 : 36$
$(2, 4)$	18	121964031	$\theta = 80000000$ ⋆	$5 : 21 : 04$
$(3, 2)$	33	206726218	74737027	$5 : 43 : 26$
$(3, 3)$	18	139943577	$\theta = 80000000$ ⋆	$5 : 18 : 57$
$(4, 2)$	21	177451069	$\theta = 80000000$ ⋆	$6 : 12 : 48$

⋆ In a "starred" experiment, we exit TLC when the number of distinct states it examines reachs a threshold θ. This is supported by a TLC nightly build as of 01-28-2019 (at 05:56).

Table 3: Model checking results of verifying that *AbsJupiter* satisfies *WLSpec*.

TLC Model ($\#Clients, \#Chars$)	**Diameter**	**# States**	**# Distinct States**	**Checking Time** ($hh : mm : ss$)
$(2, 2)$	19	50215	28307	$0 : 00 : 03$
$(2, 3)$	28	150627005	75726121	$1 : 54 : 46$
$(2, 4)$	20	153275009	$\theta = 100000000$ ⋆	$3 : 54 : 49$
$(3, 2)$	33	206726218	74737027	$2 : 46 : 02$
$(3, 3)$	25	175457016	$\theta = 100000000$ ⋆	$2 : 59 : 29$
$(4, 2)$	22	222738876	$\theta = 100000000$ ⋆	$3 : 16 : 45$

(i.e., the length of the longest behavior of protocol), the number of states TLC examines, the number of distinct states, and the checking time in $hh : mm : ss$.

6.1 Verifying Refinement Relation among *Jupiter* Protocols

We verify the refinement mapping *AbsJ* from *CJupiter* to *AbsJupiter* defined in *CJupiterImplAbsJupiter* by checking that each behavior of *CJupiter* with variables substituted by *AbsJ* is a behavior allowed by *AbsJupiter*. The model checking results are shown in Table 2. The results on verification of the refinement mappings defined in *XJupiterImplCJupiter* and *AJupiterImplXJupiter* are given in Appendix C.

6.2 Verifying Correctness of *Jupiter* Protocols

We present the model checking results of verifying that *AbsJupiter* satisfies the weak list specification *WLSpec* [8]. To express *WLSpec* in TLA$^+$, we create *AbsJupiterH* which extends *AbsJupiter* with a history variable *hlist* [18]. *AbsJupiterH* behaves exactly as *AbsJupiter*, except that it collects the new list state $list'[r]$ in each step in *hlist*. We check that *WLSpec* is an invariant of *AbsJupiterH* using TLC, and the model checking results are shown in Table 3.

7 Related Work

OT was pioneered by Ellis et al. in 1989 [10]. Though the idea of OT is simple, OT-based protocols are subtle and error-prone. For example, the dOPT protocol in [10] for P2P systems did not work in all cases [23] [26]. Remarkably, after several failed attempts [23] [12] [13] [19], it was shown impossible [22] to design OT functions (and thus OT-based protocols) for P2P systems with signatures as in *Ins* and *Del*. On the other hand, researchers made efforts to gain a better understanding why some OT-based protocols work [28] [30]. In this paper, we identify the key OT issue in centralized settings, present a generic solution, and summarize several techniques to carry out the solution. We also propose *AbsJupiter*, inspired by the COT protocol for P2P systems [28], which is abstract from implementation and captures the essence of existing *Jupiter* protocols.

The first *Jupiter* protocol appeared in 1995 [21] and is now used in many collaborative editors such as Google Docs [7], Firepad [1], and SubEthaEdit [6]. However, its original description involves only a single client. Based on the notion of COT they developed before [28], Xu et al. [30] have reported a multi-client version of *Jupiter*, which we call *XJupiter*. *XJupiter* uses $2D$ digraphs to manage COTs. Independently, Attiya et al. [9] described another multi-client version of *Jupiter*, which we call *AJupiter*. *AJupiter* relies on the acknowledge mechanism and uses $1D$ buffers to manage OTs, thus reducing the metadata overhead. To facilitate the proof that *XJupiter* satisfies the weak list specification [8], Wei et al. [29] have proposed *CJupiter* (Compact Jupiter), which is equivalent to *XJupiter*. *CJupiter* is compact in the sense that at a high level, it maintains only a single n-ary digraph that encompasses all replica states. In this work, we have established the (data) refinement relation [11] [18] [17] among the family of *Jupiter* protocols, including *AbsJupiter*, *CJupiter*, *XJupiter*, and *AJupiter*.

Many work have been devoted to formal verification of OT functions for lists or trees [13] [22] [20] [27] [25]. In contrast, little has been done on formal verification of complete OT-based protocols. To our knowledge, we are the first to formally specify and verify a family of OT-based *Jupiter* protocols and the refinement relation among them. This is conducted using TLA$^+$ [16] and TLC [31].

8 Conclusion

We study a family of *Jupiter* protocols for replicated lists. We have identified the key OT issue of *Jupiter* and presented a generic solution. We summarized several techniques for carrying out the solution and proposed an implementation-independent *AbsJupiter* protocol. We have also established the (data) refinement relation among *Jupiter* protocols. These protocols and the refinement relation among them have been formally specified and verified using TLA$^+$ and TLC. We are now working on the mechanical correctness proofs for these protocols with respect to desired properties and the refinement relation among them using TLAPS [4], a proof system for TLA$^+$.

References

1. Firepad. https://firepad.io/
2. Google docs. https://docs.google.com
3. Jupiter-Refinement-Project in tla+. https://github.com/hengxin/jupiter-refinement-project.git
4. Microsoft ResearchInria Joint Centre: TLA+ Proof System (TLAPS). https://tla.msr-inria.inria.fr/tlaps/content/Home.html
5. Overleaf. https://www.overleaf.com/
6. Subethaedit. https://subethaedit.net/
7. What's different about the new google docs: Making collaboration fast. https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html
8. Attiya, H., Burckhardt, S., Gotsman, A., Morrison, A., Yang, H., Zawirski, M.: Specification and complexity of collaborative text editing. In: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing. pp. 259–268. PODC '16 (2016). https://doi.org/10.1145/2933057.2933090
9. Attiya, H., Gotsman, A.: personal communication
10. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. In: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data. pp. 399–407. SIGMOD '89 (1989). https://doi.org/10.1145/67544.66963
11. Hoare, C.A.: Proof of correctness of data representations. Acta Inf. **1**(4), 271–281 (Dec 1972). https://doi.org/10.1007/BF00289507
12. Imine, A., Molli, P., Oster, G., Rusinowitch, M.: Proving correctness of transformation functions in real-time groupware. In: Proceedings of the Eighth European Conference on Computer Supported Cooperative Work. pp. 277–293. ECSCW '03 (2003), http://www.ecscw.org/2003/015Imine_ecscw03.pdf
13. Imine, A., Rusinowitch, M., Oster, G., Molli, P.: Formal design and verification of operational transformation algorithms for copies convergence. Theor. Comput. Sci. **351**(2), 167–183 (Feb 2006). https://doi.org/10.1016/j.tcs.2005.09.066
14. Lamport, L.: Summary of tla+. http://lamport.azurewebsites.net/tla/summary-standalone.pdf
15. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (May 1994). https://doi.org/10.1145/177492.177726
16. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
17. Lamport, L.: If you're not writing a program, don't use a programming language. Bulletin of the EATCS **125** (2018), http://eatcs.org/beatcs/index.php/beatcs/article/view/539
18. Lamport, L., Merz, S.: Auxiliary variables in tla+. CoRR **abs/1703.05121** (2017), http://arxiv.org/abs/1703.05121
19. Li, D., Li, R.: An approach to ensuring consistency in peer-to-peer real-time group editors. Computer Supported Cooperative Work (CSCW) **17**(5), 553–611 (Dec 2008). https://doi.org/10.1007/s10606-005-9009-5
20. Liu, Y., Xu, Y., Zhang, S.J., Sun, C.: Formal verification of operational transformation. In: FM 2014: Formal Methods. pp. 432–448 (2014). https://doi.org/10.1007/978-3-319-06410-9_30
21. Nichols, D.A., Curtis, P., Dixon, M., Lamping, J.: High-latency, low-bandwidth windowing in the jupiter collaboration system. In: Proceedings of the 8th Annual

ACM Symposium on User Interface and Software Technology. pp. 111–120. UIST '95 (1995). https://doi.org/10.1145/215585.215706
22. Randolph, A., Boucheneb, H., Imine, A., Quintero, A.: On consistency of operational transformation approach. In: Proceedings 14th International Workshop on Verification of Infinite-State Systems, Infinity 2012, Paris, France, 27th August 2012. pp. 45–59 (2012). https://doi.org/10.4204/EPTCS.107.5
23. Ressel, M., Nitsche-Ruhland, D., Gunzenhäuser, R.: 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. pp. 288–297. CSCW '96 (1996). https://doi.org/10.1145/240080.240305
24. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems. pp. 386–400. SSS'11 (2011). https://doi.org/10.1007/978-3-642-24550-3_29
25. Sinchuk, S., Chuprikov, P., Solomatov, K.: Verified operational transformation for trees. In: Interactive Theorem Proving. pp. 358–373 (2016). https://doi.org/10.1007/978-3-319-43144-4_22
26. Sun, C., Ellis, C.: Operational transformation in real-time group editors: Issues, algorithms, and achievements. In: Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work. pp. 59–68. CSCW '98 (1998). https://doi.org/10.1145/289444.289469
27. Sun, C., Xu, Y., Agustina, A.: Exhaustive search of puzzles in operational transformation. In: Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work. pp. 519–529. CSCW '14 (2014). https://doi.org/10.1145/2531602.2531630
28. Sun, D., Sun, C.: Context-based operational transformation in distributed collaborative editing systems. IEEE Trans. Parallel Distrib. Syst. **20**(10), 1454–1470 (Oct 2009). https://doi.org/10.1109/TPDS.2008.240
29. Wei, H., Huang, Y., Lu, J.: Specification and implementation of replicated list: The jupiter protocol revisited. In: 22nd International Conference on Principles of Distributed Systems (OPODIS 2018). vol. 125, pp. 12:1–12:16 (2018). https://doi.org/10.4230/LIPIcs.OPODIS.2018.12
30. Xu, Y., Sun, C., Li, M.: Achieving convergence in operational transformation: Conditions, mechanisms and systems. In: Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work. pp. 505–518. CSCW '14 (2014). https://doi.org/10.1145/2531602.2531629
31. Yu, Y., Manolios, P., Lamport, L.: Model checking tla+ specifications. In: Correct Hardware Design and Verification Methods. pp. 54–66 (1999). https://doi.org/10.1007/3-540-48153-2_6
32. Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G.R., Zhao, X., Zhang, Y., Jain, P.U., Stumm, M.: Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. pp. 249–265. OSDI'14 (2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

Appendix A Preliminaries on TLA$^+$

The specification language TLA$^+$ was designed by Lamport for modelling and reasoning about concurrent and distributed programs [16]. In TLA$^+$, systems are modelled as state machines. A state machine is described by its initial states and actions. A *state* is an assignment of values to variables. An *action* is a relation between old states and new states, and is represented by a formula over unprimed variables referring to the old state and the primed variables referring to the new state. For example, $x' = y + 42$ is the relation asserting that the value of x in the new state is 42 greater than that of y in the old state.

TLA$^+$ is based on TLA, the Temporal Logic of Actions [15]. A program is specified in TLA$^+$ as a temporal formula of TLA of the form $\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land L$, where *Init* is a predicate specifying all possible initial states of the program, *Next* specifies the next-state relation of the program, $\Box$ is the temporal operator read *always*, *vars* is the tuple of all variables used in the program, and L is a liveness property. The next-state relation *Next* is typically a disjunction of all the actions of the program. The expression $[\mathit{Next}]_{\mathit{vars}}$ is true if either *Next* is true, meaning that some action is true and thus taken, or *vars* stutters, meaning that their values are unchanged.

TLA$^+$ combines TLA with first-order logic and ZF set theory. Table A.1 summarizes the operators in logic and set theory we use in this paper. It is an excerpt from the complete summary of TLA$^+$ [14] and shows only the operators that have special notations in TLA$^+$. Note that the UNCHANGED clauses are often omitted in text due to space limit.

Specifications of programs are grouped into *modules*. In a module, we can declare constants (CONSTANTS) and variables (VARIABLES), define operators ($F(x_1, \cdots, x_n) \triangleq \mathit{exp}$), and claim theorems (THEOREM P). A module M can import the declarations, definitions, and theorems from other modules $M_1, \cdots, M_n$ by extending them, namely writing EXTENDS $M_1, \cdots, M_n$ in M. Modules can also be instantiated. Consider the following INSTANCE statement in module M:

$$\mathit{IM}_1 \triangleq \text{INSTANCE } M_1 \text{ WITH } p_1 \leftarrow e_1, \cdots, p_n \leftarrow e_n$$

where p_i consist of all declared constants and variables of M_1 and e_i are valid expressions in M. For each operator F and its definition d of module M_1, this defines F to be the operator, denoted $\mathit{IM}_1!F$, whose definition is obtained from d by replacing each p_i of M_1 with e_i. In addition, the implicit substitution rule in TLA$^+$ allows us to drop any substitution $p_j \leftarrow e_j$ if e_j is the same with p_j.

TLC is an explicit-state model checker for TLA$^+$ [31]. It can compute and explore the state space of finite-state instances of TLA$^+$ specifications. These finite-state instances are called TLC models of TLA$^+$ specifications. For example, a TLC model of a specification describing a distributed system consisting of a set of processors declared as CONSTANTS *Proc* should instantiate *Proc* with a set consisting of a fixed number of processors, like $\mathit{Proc} \triangleq \{1, 2, 3\}$. We can also represent a process by a TLC model value, which is considered to be unequal to any other values in TLA$^+$. So, we can instantiate *Proc* with a set of model values

Proc $\triangleq \{p1, p2, p3\}$. Moreover, if permuting the elements in a set of model values does not change whether or not a behavior satisfies a desired specification, we can further use the Symmetry Set technique to reduce the state space that TLC has to check [16].

In TLA$^+$, *refinement* is logical implication. Suppose we have two specifications: *AbsSpec* defined in module *AbsModule* with variables $x_1, \cdots, x_m, y_1, \cdots, y_n$, and *ImplSpec* defined in module *ImplModule* with variables $x_1, \cdots, x_m, z_1, \cdots, z_p$. Let X, Y, and Z denote $x_1, \cdots, x_m$, $y_1, \cdots, y_n$, and $z_1, \cdots, z_p$, respectively. To verify that *ImplSpec* refines (a.k.a implements) *AbsSpec*, formally *ImplSpec* $\Rightarrow$ *AbsSpec*, we need to show that for each behavior satisfying *ImplSpec*, there is some way to assign values of the variables Y in each state so that the resulting behavior satisfies *AbsSpec* [18]. This can be done by explicitly specifying those values of Y in terms of X and Z. Specifically, for each y_i, we define an expression $\overline{y_i}$ in terms of X and Z, substitute $y_i \leftarrow \overline{y_i}$ in *AbsSpec* to get $\overline{AbsSpec}$, and show that *ImplSpec* implements $\overline{AbsSpec}$. The substitution $y_i \leftarrow \overline{y_i}$ is called a *refinement mapping*. To verify the assertion that *ImplSpec* refines/implements *AbsSpec* under such a refinement mapping in TLA$^+$, we can add the following definition to module *ImplModule* (*AbsSub* is a fresh identifier),

$$AbsSub \triangleq \textsc{instance}\ AbsModule\ \textsc{with}\ y_1 \leftarrow \overline{y_1}, \cdots, y_n \leftarrow \overline{y_n}$$

and let TLC check the theorem THEOREM *ImplSpec* $\Rightarrow$ *AbsSub*!*AbsSpec* added to module *ImplModule*.

Table A.1: A summary of TLA$^+$ operators used in this paper.

	Operators	**Meaning**
Logic	$\text{CHOOSE } x \in S : p$	An x in S satisfying p
Sets	$\text{SUBSET } S$	Powerset (i.e., set of subsets) of S
	$\{e : x \in S\}$	Set of elements e such that x is in S
	$\{x \in S : p\}$	Set of elements x in S satisfying p
Functions	$f[e]$	Function application
	$[x \in S \mapsto e]$	Function f such that $f[x] = e$ for $x \in S$
	$[S \to T]$	Set of functions mapping from S to T
	$[f \text{ EXCEPT } ![e_1] = e_2]$	Function $\hat{f}$ equal to f except that $\hat{f}[e_1] = e_2$
	$[f \text{ EXCEPT } ![c] = e]$, where e contains @	@ in e stands for $f[c]$
Records	$e.h$	The h-field of record e
	$[h_1 \mapsto e_1, \cdots, h_n \mapsto e_n]$	The record whose h_i field is e_i
	$[h_1 : S_1, \cdots, h_n : S_n]$	Set of all records with h_i field in S_i
	$[r \text{ EXCEPT } !.h = e]$	Record $\hat{r}$ equal to r except that $\hat{r}.h = e$
	$[r \text{ EXCEPT } !.h = e]$, where e contains @	@ in e stands for $r.h$
Tuples	$e[i]$	The i^{th} component of tuple e
	$\langle e_1, \cdots, \rangle$	The n-tuple whose i^{th} component is e_i
	$S_1 \times \cdots \times S_n$	The set of all n-tuples with i^{th} component in S_i
Sequences	$Seq(S)$	The set of all sequences of elements of the set S
	$Head(s)$	The first element of sequence s
	$Last(s)$	The last element of sequence s
	$Tail(s)$	The tail of sequence s, which consists of s with its head removed
Action Operators	e'	The value of e in the new state of an action
	$\text{UNCHANGED } e$	$e' = e$
	$[A]_e$	$A \lor (e' = e)$
Temporal Operators	$\Box F$	F is always true

Appendix B TLA$^+$ Code

```
MODULE OT

OTII(lins, rins) ≜   lins is transformed against rins; II is for Ins vs. Ins.
    IF lins.pos < rins.pos THEN lins
     ELSE IF lins.pos > rins.pos
             THEN [lins EXCEPT !.pos = @ + 1]
             ELSE IF lins.ch = rins.ch THEN Nop
                     ELSE IF lins.pr > rins.pr THEN lins  using "priority"
                             ELSE [lins EXCEPT !.pos = @ + 1]
OTID(ins, del) ≜  ins is transformed against del
    IF ins.pos ≤ del.pos THEN ins
                          ELSE [ins EXCEPT !.pos = @ − 1]
OTDI(del, ins) ≜  del is transformed against ins
    IF del.pos < ins.pos THEN del
                          ELSE [del EXCEPT !.pos = @ + 1]
OTDD(ldel, rdel) ≜  ldel is transformed against rdel; DD is for Del vs. Del.
    IF ldel.pos < rdel.pos THEN ldel
     ELSE IF ldel.pos = rdel.pos THEN Nop
             ELSE [ldel EXCEPT !.pos = @ − 1]
OT(lop, rop) ≜  lop is transformed against rop
    CASE lop = Nop ∨ rop = Nop → lop
      □ lop.type = "Ins" ∧ rop.type = "Ins" → OTII(lop, rop)
      □ lop.type = "Ins" ∧ rop.type = "Del" → OTID(lop, rop)
      □ lop.type = "Del" ∧ rop.type = "Ins" → OTDI(lop, rop)
      □ lop.type = "Del" ∧ rop.type = "Del" → OTDD(lop, rop)
```

```
MODULE WLSpec

Compatible(l1, l2) ≜  Are l₁ and l₂ compatible?
    ∨ seq1 = seq2  Obviously true
    ∨ LET commonElements ≜ Range(l1) ∩ Range(l2)
       IN  ∀ e1, e2 ∈ commonElements :
              ∨ e1 = e2
              ∨ FirstIndexOfElement(l1, e1) < FirstIndexOfElement(l1, e2)
                ≡ FirstIndexOfElement(l2, e1) < FirstIndexOfElement(l2, e2)
```

```
MODULE SV

so(oid1, oid2, sv) ≜  Is oid1 totally ordered before oid2 according to sv?
   LET pos1 ≜ FirstIndexOfElementSafe(sv, oid1)  0 if oid1 is not in sv
       pos2 ≜ FirstIndexOfElementSafe(sv, oid2)  0 if oid2 is not in sv
   IN  IF pos1 ≠ 0 ∧ pos2 ≠ 0  CASE (1): both have been at the server
        THEN pos1 < pos2        using the order they reach the server
        ELSE IF pos1 = 0 ∧ pos2 = 0       CASE (2): none have been at the server
                THEN oid1.seq < oid2.seq  using the order they are generated
                ELSE pos1 ≠ 0  CASE (3): the one has been at the server is ahead
```

MODULE *Digraph*

$xForm(NextEdge(\_, \_, \_), r, cop, g) \triangleq$ Transform cop in g at replica r; see Figure 4.

$\quad$ LET $u \triangleq$ CHOOSE $n \in g.node : n = cop.ctx \quad v \triangleq u \cup \{cop.oid\}$

$\quad\quad xFormHelper(uh, vh, coph, gh) \triangleq$ gh: extra digraph produced in $xForm$

$\quad\quad\quad$ IF $uh = ds[r]$ THEN $[xcop \mapsto coph, xg \mapsto gh,$

$\quad\quad\quad\quad lg \mapsto [node \mapsto \{vh\}, edge \mapsto \{[from \mapsto uh, to \mapsto vh, cop \mapsto coph]\}]]$

$\quad\quad\quad$ ELSE LET $e \triangleq NextEdge(r, uh, g)$ specific to *CJupiter* and *XJupiter*

$\quad\quad\quad\quad ecop \triangleq e.cop \quad eu \triangleq e.to \quad ev \triangleq vh \cup \{ecop.oid\}$

$\quad\quad\quad\quad coph2ecop \triangleq COT(coph, ecop) \quad ecop2coph \triangleq COT(ecop, coph)$

$\quad\quad\quad\quad$ IN $xFormHelper(eu, ev, coph2ecop,$

$\quad\quad\quad\quad\quad gh \oplus [node \mapsto \{ev\},$ union with new $node$ and $edge$

$\quad\quad\quad\quad\quad\quad edge \mapsto \{[from \mapsto vh, to \mapsto ev, cop \mapsto ecop2coph],$

$\quad\quad\quad\quad\quad\quad\quad [from \mapsto eu, to \mapsto ev, cop \mapsto coph2ecop]\}])$

$\quad$ IN $xFormHelper(u, v, cop, [node \mapsto \{v\}, edge \mapsto \{[from \mapsto u, to \mapsto v, cop \mapsto cop]\}])$

MODULE *CJupiter*

VARIABLES css $\quad css[r]$: the n-ary digraph maintained at replica r

$NextEdge(r, u, g) \triangleq$ CHOOSE $e \in g.edge : \land\ e.from = u$

$\quad\quad\quad \land\ \forall ue \in g.edge \setminus \{e\} :$

$\quad\quad\quad\quad (ue.from = u) \Rightarrow so(e.cop.oid, ue.cop.oid, serial[r])$

$Perform(r, cop) \triangleq$ LET $xform \triangleq xForm(NextEdge, r, cop, css[r])$

$\quad$ IN $\land\ css' = [css$ EXCEPT $![r] = @ \oplus xform.xg]$

$\quad\quad \land$ apply $xform.xcop.op$ to $list[r]$

$Do(c, op) \triangleq$ LET $cop \triangleq [op \mapsto op, oid \mapsto [c \mapsto c, seq \mapsto cseq[c]], ctx \mapsto ds[c]]$

$\quad u \triangleq ds[c] \quad v \triangleq u \cup \{cop.oid\}$

$\quad$ IN $\land\ css' = [css$ EXCEPT $![c] =$ append cop to $u \triangleq ds[c]$

$\quad\quad @ \oplus [node \mapsto \{v\}, edge \mapsto \{[from \mapsto u, to \mapsto v, cop \mapsto cop]\}]]$

$\quad \land$ apply op to $list[c]$; send cop to the *Server*

$Rev(c, cop), \quad SRev(cop)$ the same with those in *AbsJupiter*

MODULE *AJupiter*

$$
\begin{array}{l}
\textsc{variables}\ \mathit{cbuf},\ \mathit{crec},\ \mathit{sbuf},\ \mathit{srec}\\
\mathit{AJMsg} \triangleq [c : \mathit{Client},\ \mathit{ack} : \mathit{Nat},\ \mathit{op} : \mathit{Op} \cup \{\mathit{Nop}\}] \cup \quad \text{from client } c \text{ to } \mathit{Server}\\
\qquad\qquad [\mathit{ack} : \mathit{Nat},\ \mathit{op} : \mathit{Op} \cup \{\mathit{Nop}\}] \quad \text{from } \mathit{Server} \text{ to clients}
\end{array}
$$

$$
\begin{array}{l}
\mathit{Do}(c,\ \mathit{op}) \triangleq \begin{array}[t]{l}
\land\ \mathit{cbuf}' = [\mathit{cbuf}\ \textsc{except}\ ![c] = \mathit{Append}(@,\ \mathit{op})]\\
\land\ \mathit{crec}' = [\mathit{crec}\ \textsc{except}\ ![c] = 0]\\
\land\ \text{apply } \mathit{op} \text{ to } \mathit{list}[c]\\
\land\ \text{send } [c \mapsto c, \mathit{ack} \mapsto \mathit{crec}[c], \mathit{op} \mapsto \mathit{op}] \text{ to the } \mathit{Server}
\end{array}\\
\mathit{Rev}(c,\ m) \triangleq \textsc{let}\ \mathit{xform} \triangleq \mathit{xFormShift}(m.\mathit{op},\ \mathit{cbuf}[c],\ m.\mathit{ack}) \quad \mathit{xform} : [\mathit{xop}, \mathit{xops}]\\
\qquad\qquad \textsc{in}\ \begin{array}[t]{l}
\land\ \mathit{cbuf}' = [\mathit{cbuf}\ \textsc{except}\ ![c] = \mathit{xform}.\mathit{xops}]\\
\land\ \mathit{crec}' = [\mathit{crec}\ \textsc{except}\ ![c] = @ + 1]\\
\land\ \text{apply } \mathit{xform}.\mathit{xop} \text{ to } \mathit{list}[c]
\end{array}\\
\mathit{SRev}(m) \triangleq \textsc{let}\ \begin{array}[t]{l}
c \triangleq m.c\\
\mathit{xform} \triangleq \mathit{xFormShift}(m.\mathit{op},\ \mathit{sbuf}[c],\ m.\mathit{ack}) \quad \mathit{xform} : [\mathit{xop}, \mathit{xops}]\\
\mathit{xop} \triangleq \mathit{xform}.\mathit{xop}
\end{array}\\
\qquad\qquad \textsc{in}\ \begin{array}[t]{l}
\land\ \mathit{srec}' = [\mathit{cl} \in \mathit{Client} \mapsto \textsc{if}\ \mathit{cl} = c\ \textsc{then}\ \mathit{srec}[\mathit{cl}] + 1\ \textsc{else}\ 0]\\
\land\ \mathit{sbuf}' = [\mathit{cl} \in \mathit{Client} \mapsto \textsc{if}\ \mathit{cl} = c\ \textsc{then}\ \mathit{xform}.\mathit{xops}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsc{else}\ \mathit{Append}(\mathit{sbuf}[\mathit{cl}],\ \mathit{xop})]\\
\land\ \text{apply } \mathit{xop} \text{ to } \mathit{list}[\mathit{Server}]\\
\land\ \text{send } [\mathit{ack} \mapsto \mathit{srec}[\mathit{cl}], \mathit{op} \mapsto \mathit{xop}] \text{ to client } \mathit{cl} \neq c
\end{array}
\end{array}
$$

Appendix C Model Checking Results

Table C.1: Model checking results of verifying that *CJupiter* refines *AbsJupiter*.

TLC Model ($\#Clients, \#Chars$)	**Diameter**	**# States**	**# Distinct States**	**Checking Time** ($hh : mm : ss$)
$(1, 1)$	5	7	6	$0 : 00 : 00$
$(1, 2)$	9	86	57	$0 : 00 : 00$
$(1, 3)$	13	1696	1014	$0 : 00 : 01$
$(1, 4)$	17	53273	30393	$0 : 00 : 06$
$(2, 1)$	10	71	53	$0 : 00 : 01$
$(2, 2)$	19	50215	28307	$0 : 00 : 05$
$(2, 3)$	28	150627005	75726121	$4 : 37 : 36$
$(2, 4)$	18	121964031	$\theta = 80000000$ *	$5 : 21 : 04$
$(3, 1)$	17	2785	1288	$0 : 00 : 01$
$(3, 2)$	33	206726218	74737027	$5 : 43 : 26$
$(3, 3)$	18	139943577	$\theta = 80000000$ *	$5 : 18 : 57$
$(4, 1)$	26	194877	61117	$0 : 00 : 18$
$(4, 2)$	21	177451069	$\theta = 80000000$ *	$6 : 12 : 48$

Table C.2: Model checking results of verifying that *XJupiter* refines *CJupiter*.

TLC Model ($\#Clients, \#Chars$)	**Diameter**	**# States**	**# Distinct States**	**Checking Time** ($hh : mm : ss$)
$(1, 1)$	5	7	6	$0 : 00 : 00$
$(1, 2)$	9	86	57	$0 : 00 : 00$
$(1, 3)$	13	1696	1014	$0 : 00 : 01$
$(1, 4)$	17	53273	30393	$0 : 00 : 07$
$(2, 1)$	10	71	53	$0 : 00 : 00$
$(2, 2)$	19	50215	28307	$0 : 00 : 07$
$(2, 3)$	28	150627005	75726121	$5 : 38 : 00$
$(2, 4)$	19	122113291	$\theta = 80000000$ ⋆	$8 : 01 : 35$
$(3, 1)$	17	2785	1288	$0 : 00 : 02$
$(3, 2)$	33	206726218	74737027	$8 : 50 : 40$
$(3, 3)$	20	139577795	$\theta = 80000000$ ⋆	$8 : 59 : 52$
$(4, 1)$	26	194877	61117	$0 : 00 : 30$
$(4, 2)$	19	175896403	$\theta = 80000000$ ⋆	$11 : 40 : 50$

Table C.3: Model checking results of verifying that *AJupiter* refines *XJupiter*.

TLC Model ($\#Clients, \#Chars$)	**Diameter**	**# States**	**# Distinct States**	**Checking Time** ($hh : mm : ss$)
$(1, 1)$	5	7	6	$0 : 00 : 01$
$(1, 2)$	9	86	57	$0 : 00 : 01$
$(1, 3)$	13	1696	1014	$0 : 00 : 01$
$(1, 4)$	17	53273	30393	$0 : 00 : 07$
$(2, 1)$	10	71	53	$0 : 00 : 00$
$(2, 2)$	19	50215	28307	$0 : 00 : 05$
$(2, 3)$	28	150627005	75726121	$4 : 23 : 52$
$(2, 4)$	18	122137621	$\theta = 80000000$ ⋆	$3 : 52 : 46$
$(3, 1)$	17	2785	1288	$0 : 00 : 01$
$(3, 2)$	33	206726218	74737027	$4 : 52 : 39$
$(3, 3)$	18	139823551	$\theta = 80000000$ ⋆	$4 : 48 : 23$
$(4, 1)$	26	194877	61117	$0 : 00 : 17$
$(4, 2)$	21	176794063	$\theta = 80000000$ ⋆	$3 : 49 : 58$

Table C.4: Model checking results of verifying that *AbsJupiter* satisfies *WLSpec*.

TLC Model ($\#Clients, \#Chars$)	**Diameter**	**# States**	**# Distinct States**	**Checking Time** ($hh : mm : ss$)
$(1, 1)$	5	7	6	$0 : 00 : 01$
$(1, 2)$	9	86	57	$0 : 00 : 01$
$(1, 3)$	13	1696	1014	$0 : 00 : 00$
$(1, 4)$	17	53273	30393	$0 : 00 : 04$
$(2, 1)$	10	71	53	$0 : 00 : 00$
$(2, 2)$	19	50215	28307	$0 : 00 : 03$
$(2, 3)$	28	150627005	75726121	$1 : 54 : 46$
$(2, 4)$	20	153275009	$\theta = 100000000$ *	$3 : 54 : 49$
$(3, 1)$	17	2785	1288	$0 : 00 : 01$
$(3, 2)$	33	206726218	74737027	$2 : 46 : 02$
$(3, 3)$	25	175457016	$\theta = 100000000$ *	$2 : 59 : 29$
$(4, 1)$	26	194877	61117	$0 : 00 : 09$
$(4, 2)$	22	222738876	$\theta = 100000000$ *	$3 : 16 : 45$