Hampa: Solver-aided Recency-aware Replication Appendix

Xiao Li, Farzin Houshmand, and Mohsen Lesani

University of California, Riverside
{xli289, fhous001, lesani}@ucr.edu

Table of Contents

1 Use-cases 3
2 Bound Inference Example 5
3 Proofs 6
3.1 Bound Inference 6
3.2 Convergence, Integrity and Recency 11
4 Coordination Conditions 39
5 Protocol 42
6 Extra Experiment 45

1 Use-cases

Class BankAccount
$\Sigma :=$ Int b

$\mathcal{I} := \lambda b.\ b \geq 0$

deposit$(a) := 0\ \lambda b.$
$\langle a \geq 0, \quad b + a, \quad \bot \rangle$
withdraw$(a) := 0\ \lambda b.$
$\langle a \geq 0, \quad b - a, \quad \bot \rangle$
balance $:= \epsilon_1\ \lambda b.$
$\langle \mathsf{True}, \quad b, \quad b \rangle$

Fig. 1: Bank Account Use-case

Class MovieBooking

$\Sigma :=$

$\quad$ let $Rs := \mathsf{Set}\ \mathbb{N} \times \mathbb{N}$ in $\quad \triangleright$ Reservation: user identifier and movie identifier

$\quad$ let $Ms := \mathsf{Set}\ \mathbb{N} \times \mathbb{N}$ in $\quad \triangleright$ Movie: movie identifier and available space

$\quad \langle Rs, Ms \rangle$

$\mathcal{I} := \lambda \langle rs, ms \rangle .$

$\quad \mathsf{unique}\ (ms, \lambda \langle m, a \rangle .\ m) \wedge$

$\quad \mathsf{refIntegrity}\ (rs, \lambda \langle u, m \rangle .\ m, ms, \lambda \langle m, a \rangle .\ m) \wedge$

$\quad \mathsf{rowIntegrity}\ (ms, \lambda \langle m, a \rangle .\ a \geq 0)$

$\mathsf{book}(\langle u, m \rangle) := 0\ \lambda \langle rs, ms \rangle .$

$\quad \langle \langle u, m \rangle \notin rs, \quad \langle rs \cup \langle u, m \rangle,\ \mathcal{U}_{\lambda \langle m', a \rangle .\ \langle m' = m, \langle m, a-1 \rangle \rangle}\ ms \rangle, \quad \bot \rangle$

$\mathsf{cancelBook}(\langle u, m \rangle) := 0\ \lambda \langle rs, ms \rangle .$

$\quad \langle \mathsf{True}, \quad \langle rs \setminus \langle u, m \rangle,\ \mathcal{U}_{\lambda \langle m', a \rangle .\ \langle m' = m, \langle m, a+1 \rangle \rangle}\ ms \rangle, \quad \bot \rangle$

$\mathsf{offScreen}(m) := 0\ \lambda \langle rs, ms \rangle .$

$\quad \langle \mathsf{True}, \quad \langle rs,\ ms \setminus \sigma_{\lambda \langle m', a \rangle .\ m' = m}\ ms \rangle, \quad \bot \rangle$

$\mathsf{specialReserve}(\langle m, n \rangle) := 0\ \lambda \langle rs, ms \rangle .$

$\quad \langle n > 0, \quad \langle rs,\ \mathcal{U}_{\lambda \langle m', a \rangle .\ \langle m' = m, \langle m, a-n \rangle \rangle}\ ms \rangle, \quad \bot \rangle$

$\mathsf{increaseSpace}(\langle m, n \rangle) := 0\ \lambda \langle rs, ms \rangle .$

$\quad \langle n > 0, \quad \langle rs,\ \mathcal{U}_{\lambda \langle m', a \rangle .\ \langle m' = m, \langle m, a+n \rangle \rangle}\ ms \rangle, \quad \bot \rangle$

$\mathsf{querySpace}(m) := \epsilon_1\ \lambda \langle rs, ms \rangle .$

$\quad \langle \mathsf{True}, \quad \langle rs, ms \rangle, \quad \mathit{\Pi}_{\lambda \langle m', a \rangle .\ \langle a \rangle}\ (\sigma_{\lambda \langle m', a \rangle .\ m' = m}\ ms) \rangle$

$\mathsf{queryReservations}(u) := \epsilon_2\ \lambda \langle rs, ms \rangle .$

$\quad \langle \mathsf{True}, \quad \langle rs, ms \rangle, \quad \mathit{\Pi}_{\lambda \langle u', m \rangle .\ \langle m \rangle}\ (\sigma_{\lambda \langle u', m \rangle .\ u' = u}\ rs) \rangle$

$\mathsf{querySpaces}(u) := \epsilon_3\ \lambda\, \langle rs, ms \rangle$

$\quad \langle \mathsf{True}, \quad \langle rs, ms \rangle, \quad \mathit{\Pi}_{\lambda \langle u, m, m', a \rangle\ \langle m, a \rangle}\ (rs \bowtie_{\lambda \langle u, m \rangle, \langle m', a \rangle .\ m = m'}\ ms) \rangle$

Fig. 2: Movie Booking Use-case

2 Bound Inference Example

We use three different query methods in the movie use-case to illustrate the bound inference rules. In the querySpace method, it first uses selection on the ms relation to find the specific tuple associated with the movie identifier that the user provided. Then, it projects the available space field from the tuple and returns it to user. The queryReservations method follows the same pattern. It first selects all the tuples associated with the given user identifier in the rs relation. It then projects the movie identifier field form the tuples it selected and returns them. Finally, the querySpaces method first joins the relations rs and ms based on the equality of movie identifiers. Then, it projects only the movie identifier and the available space fields on the joined relation and returns them.

Now let us take a look on how the bound inference rules are used. We want to calculate the staleness of the two relations in our movie use-case: drs for rs and dms for ms. As an example, we construct the inference for the selection operation of the querySpace below. The final constraint inferred for the querySpace method is $dms \leq \epsilon_2$. Similarly, the final constraint inferred for the queryReservations method is $drs \leq \epsilon_1$. The querySpaces method uses the join operation. The join operation is actually a combination of production and selection. By the rule CPROD, we get $drs \times dms$ as the staleness of the product relation. The product relation is passed to selection rule CSEL where the staleness remains unchanged. Finally, the projection rule CPROJ leaves the staleness unchanged as well. The final constraint inferred for the querySpace method is $drs \times dms \leq \epsilon_3$.

Next, we find the solutions of inferred constrains. Let's assume that $\epsilon_1 = 3$, $\epsilon_2 = 4$, and $\epsilon_3 = 6$. The constraints are $drs \leq 3$, $dms \leq 4$ and $drs \times dms \leq 6$. We can have multiple solutions which satisfy all the three inferred constrains. For example, a simple solution is $drs = 1, dms = 1$. But this solution does not fully utilize the potential of the recency specifications. If the update frequency of rs is twice as ms, the optimum solution is $drs = 3, dms = 2$ to favor buffering updates to the relation rs rather than ms.

$$\text{CBOP}\ \frac{\dfrac{\text{CVAR}\ \dfrac{(m' \mapsto 0) \in [m \mapsto 0, ms \mapsto dms, rs \mapsto drs, m' \mapsto 0, a \mapsto 0\rangle]}{[m \mapsto 0, ms \mapsto dms, rs \mapsto drs, m' \mapsto 0, a \mapsto 0\rangle] \vdash m' \triangleright 0, \emptyset} \quad \text{CVAR}\ \dfrac{(m \mapsto 0) \in [m \mapsto 0, ms \mapsto dms, rs \mapsto drs, m' \mapsto 0, a \mapsto 0\rangle]}{[m \mapsto 0, ms \mapsto dms, rs \mapsto drs, m' \mapsto 0, a \mapsto 0\rangle] \vdash m \triangleright 0, \emptyset}}{[m \mapsto 0, ms \mapsto dms, rs \mapsto drs, m' \mapsto 0, a \mapsto 0\rangle] \vdash m' = m \triangleright 0, \emptyset \wedge \emptyset \wedge 0 = 0 \wedge 0 = 0}$$

$$\text{CSEL}\ \frac{\text{CVAR}\ \dfrac{(ms \mapsto dms) \in [m \mapsto 0, ms \mapsto dms, rs \mapsto drs, m' \mapsto 0, a \mapsto 0\rangle]}{[m \mapsto 0, ms \mapsto dms, rs \mapsto drs, m' \mapsto 0, a \mapsto 0\rangle] \vdash ms \triangleright dms, \emptyset}}{[m \mapsto 0, ms \mapsto dms, rs \mapsto drs\rangle] \vdash \sigma_{\lambda\langle m', a\rangle.\, m'=m}\, ms \triangleright dms, \emptyset \wedge \emptyset \wedge \emptyset \wedge 0 = 0 \wedge 0 = 0 \wedge 0 = 0}$$

3 Proofs

3.1 Bound Inference

Lemma 1 (Soundness of Bound Inference). *Given an object o with the state variables $\langle \sigma_1, .., \sigma_n \rangle$, if $o \triangleright C$ that is the constraints C (over the bound variables $\overline{d\sigma_i}$) are derived for o, and S is a solution for C, then for every pair of states $\sigma = \langle v_1, .., v_n \rangle$ and $\sigma^* = \langle v_1^*, .., v_n^* \rangle$, if $\overline{\Delta(v_i, v_i^*) < S(d\sigma_i)}$ then σ is sufficiently-recent for σ^*.*

Proof.
We have that
(1) $o \triangleright C$
(2) S is a solution for C
(3) $\overline{\Delta(v_i, v_i^*) < S(d\sigma_i)}$
By inversion on [1], we have
(4) $\langle \Sigma, \mathcal{I}, \overline{m} \rangle$
(5) $C = \wedge \overline{C'}$
(6) $\overline{m \triangleright C'}$
Consider an arbitrary method
(7) $\mathsf{def}\ \epsilon\ m(x)(\langle \sigma_1, .., \sigma_n \rangle)\ \langle e_g, e_u, e_r \rangle$
By Def. 12, we need to show that
(8) let v_r be $\left[\!\left[e_r[x \mapsto v]\overline{[\sigma_i \mapsto v_i]} \right]\!\right]$
(9) let v_r^* be $\left[\!\left[e_r[x \mapsto v]\overline{[\sigma_i \mapsto v_i^*]} \right]\!\right]$,
$\Delta(v_r, v_r^*) \leq \epsilon$
By inversion on [6], we have
(10) $\mathit{free}(e_r) = \{x, \sigma_1, .., \sigma_n\}$
(11) $\Gamma \vdash e_r \triangleright \delta, C''$
(12) $\Gamma = [x \mapsto 0, \sigma_1 \mapsto d\sigma_1, .., \sigma_n \mapsto d\sigma_n]$
(13) $C' = C'' \wedge \delta \leq \epsilon$
From [2], [5] and [13], we have
(14) S is a solution for C''
(15) $S(\delta) \leq \epsilon$
By Lemma 3 on [11] and [12], we have
(16) $\Gamma \vdash e_r[x \mapsto v] \triangleright \delta, C''$
From [10] and [12], we have
(17) $\mathit{free}(e_r[x \mapsto v]) = \{\sigma_1, .., \sigma_n\}$
(18) $\Gamma(\sigma_i) = d\sigma_i$
By Lemma 2 one [17], [18], [16], [14], and [3], we have
(19) $\Delta(\left[\!\left[e[x \mapsto v]\overline{[\sigma_i \mapsto v_i]} \right]\!\right],$
$\left[\!\left[e[x \mapsto v]\overline{[\sigma_i \mapsto v_i^*]} \right]\!\right])$
$\leq S(\delta)$
From [19], [8] and [9], we have
(20) $\Delta(v_r, v_r^*) \leq S(\delta)$.
From [20] and [15], we have

$\Delta(v_r, v_r^*) \leq \epsilon$.

Lemma 2. *For every expression e, delta δ and constraint C such that $free(e) \subseteq \{\sigma_1, .., \sigma_n\}$ and $\Gamma(\sigma_i) = d\sigma_i$, if $\Gamma \vdash e \triangleright \delta, C$ and S is a solution for C, then for every pair of states $\sigma = \langle v_1, .., v_n \rangle$ and $\sigma^* = \langle v_1^*, .., v_n^* \rangle$, if $\overline{\Delta(v_i, v_i^*) < S(d\sigma_i)}$ then $\Delta([\![e[\overline{\sigma_i \mapsto v_i}]]\!], [\![e[\overline{\sigma_i \mapsto v_i^*}]]\!]) \leq S(\delta)$.*

Proof.
We have that

(1) $free(e) \subseteq \{\sigma_1, .., \sigma_n\}$
(2) $\Gamma(\sigma_i) = d\sigma_i$,
(3) $\Gamma \vdash e \triangleright \delta, C$
(4) S is a solution for C
(5) $\overline{\Delta(v_i, v_i^*) < S(d\sigma_i)}$

We show that

$\Delta([\![e[\overline{\sigma_i \mapsto v_i}]]\!], [\![e[\overline{\sigma_i \mapsto v_i^*}]]\!]) \leq S(\delta)$

Induction on the derivation of $\Gamma \vdash e \triangleright \delta, C$.
Case CVALL:

Immediate from $v[\overline{\sigma_i \mapsto v_i}] = v$

Case COP:

(6) $e = e_1 \oplus e_2$
(7) $\delta = \delta_1 + \delta_2$
(8) $C = C_1 \wedge C_2$
(9) $\Gamma \vdash e_1 \triangleright \delta_1, C_1$
(10) $\Gamma \vdash e_2 \triangleright \delta_2, C_2$
(11) $\oplus \in \{+, -, \cup, \setminus\}$

From [4] and [8], we have

(12) S is a solution for C_1
(13) S is a solution for C_2

By induction hypothesis on [9], [12] and [5], we have

(14) $\Delta([\![e_1[\overline{\sigma_i \mapsto v_i}]]\!], [\![e_1[\overline{\sigma_i \mapsto v_i^*}]]\!]) \leq S(\delta_1)$

By induction hypothesis on [10], [13] and [5], we have

(15) $\Delta([\![e_2[\overline{\sigma_i \mapsto v_i}]]\!], [\![e_2[\overline{\sigma_i \mapsto v_i^*}]]\!]) \leq S(\delta_2)$

By Lemma 4, [14] and [15], we have

(16) $\Delta([\![e_1[\overline{\sigma_i \mapsto v_i}]]\!] \oplus [\![e_2[\overline{\sigma_i \mapsto v_i}]]\!],$
$[\![e_1[\overline{\sigma_i \mapsto v_i^*}]]\!] \oplus [\![e_2[\overline{\sigma_i \mapsto v_i^*}]]\!])$
$\leq S(\delta_1) + S(\delta_2)$

that by the semantics of $\oplus$ is

(17) $\Delta([\![(e_1 \oplus e_2)[\overline{\sigma_i \mapsto v_i}]]\!],$
$[\![(e_1 \oplus e_2)[\sigma_i \mapsto v_i^*]]\!])$
$\leq S(\delta_1 + \delta_2)$

From [15], [6] and [7], we have

$$\Delta(\llbracket\, e[\overline{\sigma_i \mapsto v_i}]\,\rrbracket\,,\ \llbracket\, e[\overline{\sigma_i \mapsto v_i^*}]\,\rrbracket) \leq S(\delta)$$

Case CBOp:

Similar to the case COp.

It can be shown that

$\llbracket\, e_1[\overline{\sigma_i \mapsto v_i}]\,\rrbracket = \llbracket\, e_1[\overline{\sigma_i \mapsto v_i^*}]\,\rrbracket$

$\llbracket\, e_2[\overline{\sigma_i \mapsto v_i}]\,\rrbracket = \llbracket\, e_2[\overline{\sigma_i \mapsto v_i^*}]\,\rrbracket$

Case CVar:

(18) $e = x$

(19) $(x \mapsto \delta) \in \Gamma$

(20) $C = \emptyset$

From [1], [2], [18] and [19], we have

(21) $e = x = \sigma_i$

(22) $\delta = d\sigma_i$

From [21], we have

(23) $e[\overline{\sigma_i \mapsto v_i}] = v_i$

(24) $e[\overline{\sigma_i \mapsto v_i^*}] = v_i^*$

From [23] and [24], we have

(25) $\Delta(\llbracket\, e[\overline{\sigma_i \mapsto v_i}]\,\rrbracket\,, \llbracket\, e[\overline{\sigma_i \mapsto v_i^*}]\,\rrbracket) = \Delta(v_i, v_i^*)$

From [25] and [5], we have

(26) $\Delta(\llbracket\, e[\overline{\sigma_i \mapsto v_i}]\,\rrbracket\,, \llbracket\, e[\overline{\sigma_i \mapsto v_i^*}]\,\rrbracket) \leq S(\delta)$

Case CSel:

(27) $e = \sigma_{\lambda\langle\overline{x}\rangle.e'}(e'')$

(28) $C = C' \wedge C \wedge (\delta' = 0)$

(29) $\Gamma[\overline{x \mapsto 0}] \vdash e' \triangleright \delta', C'$

(30) $\Gamma \vdash e'' \triangleright \delta, C$

From [4] and [28], we have

(31) S is a solution for C'

(32) S is a solution for C

(33) $S(\delta') = 0$

From Lemma 3 on [29], we have

(34) For all $\overline{v}$, $\Gamma[\overline{x \mapsto 0}] \vdash e'[\overline{x \mapsto v}] \triangleright \delta', C'$

By induction hypothesis on [29], [31] and [5], we have

(35) For all $\overline{v}$,

$$\begin{aligned}&\Delta(\llbracket\, e'[\overline{\sigma_i \mapsto v_i}][\overline{x \mapsto v}]\,\rrbracket\,,\\ &\quad\llbracket\, e'[\overline{\sigma_i \mapsto v_i^*}][\overline{x \mapsto v}]\,\rrbracket)\\ &\quad\leq S(\delta')\end{aligned}$$

From [35] and [33], we have

(36) For all $\overline{v}$,

$$\begin{aligned}&\Delta(\llbracket\, e'[\overline{\sigma_i \mapsto v_i}][\overline{x \mapsto v}]\,\rrbracket\,,\\ &\quad\llbracket\, e'[\overline{\sigma_i \mapsto v_i^*}][\overline{x \mapsto v}]\,\rrbracket)\\ &\quad\leq 0\end{aligned}$$

By induction hypothesis on [30], [32] and [5], we have

(37) $\Delta([\![e''[\overline{\sigma_i \mapsto v_i}]]\!], [\![e''[\overline{\sigma_i \mapsto v_i^*}]]\!]) \leq S(\delta)$

Let

(38) $e_\phi = e'[\overline{\sigma_i \mapsto v_i}]$

(39) $e_\phi^* = e'[\overline{\sigma_i \mapsto v_i^*}]$

(40) $e_R = e''[\overline{\sigma_i \mapsto v_i}]$

(41) $e_R^* = e''[\overline{\sigma_i \mapsto v_i^*}]$

From [36] and [37], and [38]-[41], we have

(42) For all $\overline{v}$, $\Delta([\![e_\phi[\overline{x \mapsto v}]]\!], \left[\!\!\left[e_\phi^*[\overline{x \mapsto v}] \right]\!\!\right]) \leq 0$

(43) $\Delta([\![e_R]\!], [\![e_R^*]\!]) \leq S(\delta)$

From [42], we have

(44) For all $\overline{v}$, $[\![e_\phi[\overline{x \mapsto v}]]\!] = \left[\!\!\left[e_\phi^*[\overline{x \mapsto v}] \right]\!\!\right]$

By definition

(45) $[\![\sigma_{\lambda \overline{x}.e_\pi}(e_R)]\!] = \{t \mid t \in [\![e_R]\!] \ \wedge \ [\![e_\pi[\overline{x} \mapsto t]]\!] = \mathsf{true}\}$

(46) $[\![\sigma_{\lambda \overline{x}.e_\pi^*}(e_R^*)]\!] = \{t \mid t \in [\![e_R^*]\!] \ \wedge \ [\![e_\pi^*[\overline{x} \mapsto t]]\!] = \mathsf{true}\}$

From [45], [46], [43] and [44], we have

(47) $\Delta([\![\sigma_{\lambda \langle \overline{x} \rangle.e_\pi}(e_R)]\!], [\![\sigma_{\lambda \langle \overline{x} \rangle.e_\pi^*}(e_R^*)]\!]) \leq S(\delta)$

From [47], [27], [38]-[41], we have

$\Delta([\![e[\overline{\sigma_i \mapsto v_i}]]\!], [\![e[\overline{\sigma_i \mapsto v_i^*}]]\!]) \leq S(\delta)$

Case CPROJ:

Similar to case CSEL.

Case CPROD:

Similar to case COP using Lemma 5.

Lemma 3 (Helper lemma 1). *For all Γ, e, δ and C, if $\Gamma \vdash e \triangleright \delta, C$ and $\Gamma(x) = 0$ then $\Gamma \vdash e[x \mapsto v] \triangleright \delta, C$*

Proof.
Immediate by induction on $\Gamma \vdash e \triangleright \delta, C$.
The only interesting case is CVAR that is proved using CVAL.

Lemma 4 (Helper lemma 2). *For all v_1, v_2, v_1^* and v_2^*, $\Delta(v_1 \oplus v_2, v_1^* \oplus v_2^*) \leq \Delta(v_1, v_1^*) + \Delta(v_2, v_2^*)$*

Proof.
Immediate from the definition of Δ and case analysis on the two value types numbers n and relations R.

Lemma 5 (Helper lemma 3). *For all R_1, R_2, R_1^* and R_2^*,* $\Delta(R_1 \times R_2, R_1^* \times R_2^*) \leq \Delta(R_1, R_1^*) \times \Delta(R_2, R_2^*)$

Proof.
Immediate from the definition of Δ on relations R.

3.2 Convergence, Integrity and Recency

Definition 1 (Execution Context). *An execution context* $\mathbf{c}$ *is the record* $\langle \mathsf{orig}_{\mathbf{c}}, \mathsf{call}_{\mathbf{c}} \rangle$ *where* $\mathsf{orig}_{\mathbf{c}}$ *is a function from* R *to replicas* $\mathcal{N}$*, and* $\mathsf{call}_{\mathbf{c}}$ *is a function from* R *to method calls.*

Definition 2 (Execution). *In a context* $\mathbf{c}$*, an execution* x *of a set of requests* $R \subseteq R$ *is a bijective from positions* $[0..|R|-1]$ *to* R*.*

We denote the range of x *as* $\mathcal{R}(x)$*. An execution* x *of* R *defines the total order* $\prec_x$ *on* R*. A request* r *precedes another request* r' *in an execution* x *written as* $r \prec_x r'$ *iff* $x^{-1}(r) < x^{-1}(r')$*.*

Definition 3 (Replicated Execution). *In a context* $\mathbf{c}$*, a replicated execution* xs *is a function from replicas* $\mathcal{N}$ *to executions of requests* $R \subseteq R$ *such that (1) let the execution order* $\prec_{xs}$ *on* $\mathcal{N} \times R$ *be defined as: for every replica* n *and pair of requests* r *and* r'*,* $(n, r) \prec_{xs} (n, r')$ *iff* $r \prec_{xs(n)} r'$*, (2) let the visibility relation* $\rightsquigarrow_{xs}$ *on* $\mathcal{N} \times R$ *be defined as: for every request* r*, for every replica* n*,* $(\mathsf{orig}_{\mathbf{c}}(r), r) \rightsquigarrow_{xs} (n, r)$ *iff* $n \neq \mathsf{orig}_{\mathbf{c}}(r)$*, (3) let the happens-before relation* hb_{xs} *be* $(\prec_{xs} \cup \rightsquigarrow_{xs})^*$ *then,* hb_{xs} *is acyclic.*

Definition 4 (Complete Replicated Execution). *In a context* $\mathbf{c}$*, a complete replicated execution* xs *is a replicated execution where the range of* $xs(n)$ *for all replica* n *is the same.*

Definition 5 (Execution Fragment). *The execution fragment of* x *at positions* $[i..j]$ *written as* $x[i..j]$ *is the function* $\lambda k.\ x(k+i)$ *from positions* $[0..j-i]$ *to requests* $x(i), .., x(j)$*.*

Definition 6 (Execution Projection). *The projection of an execution* x *to a subset of requests* R *written as* $x|_R$ *is the bijective function from positions* $[0..|R|-1]$ *to* R *such that for every* r *and* r' *in* R*, if* $r \prec_x r'$*, then* $r \prec_{x|_R} r'$*.*

Definition 7 (Execution Concatenation). *Given two executions* x *and* x'*, if* $\mathcal{R}(x) \cap \mathcal{R}(x') = \emptyset$*, then* $x \cdot x'$ *is the bijective function from* $[0..|\mathcal{R}(x) \cup \mathcal{R}(x')| - 1]$ *to* $\mathcal{R}(x) \cup \mathcal{R}(x')$*, such that for every* r *and* r' *in* $\mathcal{R}(x)$*, if* $r \prec_x r'$*, then* $r \prec_{x \cdot x'} r'$ *and similarly for* x'*, and for every* r *in* $\mathcal{R}(x)$ *and* r' *in* $\mathcal{R}(x')$*,* $r \prec_{x \cdot x'} r'$*.*

Definition 8 (State). *In a context* $\mathbf{c}$*, the state function* $\mathbf{s}$ *of an execution* x *is a function from positions* $[0..|\mathcal{R}(x)|]$ *to states* Σ *such that* $\mathbf{s}(0) = \sigma_0$ *and for every* $0 \leq i < |\mathcal{R}(x)|$*,* $\mathbf{s}(i+1) = \mathsf{update}(\mathsf{call}_{\mathbf{c}}(x(i)))(\mathbf{s}(i))$*. The state function is lifted to replicated executions. The state function* $\mathbf{ss}$ *of a replicated execution* xs *is a function from replicas* n *in* $\mathcal{N}$ *to the state function of the execution* $xs(n)$*.*

Definition 9 (Convergent). *A complete replicated execution* xs *of a context* $\mathbf{c}$ *is convergent iff for every pair of replicas* n *and* n'*,* $\mathbf{ss}(n)(|R|) = \mathbf{ss}(n')(|R|)$ *where* $\mathbf{ss}$ *is the state function of* xs*.*

Definition 10 (Execution with Integrity). *In a context* $\mathbf{c}$*, a request* r *has integrity in an execution* x *written as* $\mathit{integrity}(\mathbf{c}, x, r)$ *iff* $\mathit{integrity}(\mathbf{s}(i), \mathit{call}_{\mathbf{c}}(r))$ *where* $\mathbf{s}$ *is the state function of* x*. In a context* $\mathbf{c}$*, an execution* x *has integrity written as* $\mathit{integrity}(\mathbf{c}, x)$ *iff every request* r *in* $\mathcal{R}(x)$ *in* x *has integrity. A replicated execution* xs *of a context* $\mathbf{c}$ *has integrity written as* $\mathit{integrity}(\mathbf{c}, xs)$ *iff for every replica* n*, the execution* $xs(n)$ *has integrity.*

Definition 11 (Permissible Execution). *In a context* $\mathbf{c}$*, a request* r *in an execution* x *is permissible* $\mathcal{P}(\mathbf{c}, x, r)$ *iff,* $\mathcal{P}(\mathbf{s}(i),\ \mathit{call}_{\mathbf{c}}(r))$ *where* $\mathbf{s}$ *is the state function of* x*, and* i *is* $x^{-1}(r)$*, In a context* $\mathbf{c}$*, an execution* x *is permissible* $\mathcal{P}(\mathbf{c}, x)$ *iff every request* r *in* $\mathcal{R}(x)$ *is permissible in* x*. A replicated execution* xs *of a context* $\mathbf{c}$ *is permissible* $\mathcal{P}(\mathbf{c}, xs)$ *iff the execution* $xs(n)$ *of every node* n *is permissible.*

Definition 12 (Locally-permissible replicated execution). *A replicated execution* xs *of a context* $\mathbf{c}$ *is locally-permissible iff for every request* r *in* xs*,* $\mathcal{P}(\mathbf{s}(i), r)$ *where* x *is* $xs(\mathit{orig}_{\mathbf{c}}(r))$*,* $\mathbf{s}$ *is the state function of* x *and* $i = x^{-1}(r)$*;*

Definition 13 (Conflict-synchronizing). *A replicated execution* xs *of a context* $\mathbf{c}$ *is conflict-synchronizing iff for every pair of requests* r *and* r' *such that* $\mathit{call}_c(r) \bowtie \mathit{call}_c(r')$*,*

- *if* $r \in xs(n)$ *and* $r' \in xs(n')$ *then* $r' \in xs(n)$ *or* $r \in xs(n')$*.*
- *if* $r \prec_{xs(n)} r'$*,* $r' \in xs(n')$*, then* $r \in xs(n')$*.*
- *if* $r \prec_{xs(n)} r'$ *and* $r, r' \in xs(n')$*, then* $r \prec_{xs(n')} r'$*.*

Definition 14 (State-conflict-synchronizing). *Similar to Def. 13 where conflict* $\bowtie$ *is replaced with* $\mathcal{S}$*-conflict* $\bowtie_{\mathcal{S}}$*.*

Definition 15 (Permissible-conflict-synchronizing). *Similar to Def. 13 where conflict* $\bowtie$ *is replaced with* $\mathcal{P}$*-conflict* $\bowtie_{\mathcal{P}}$*.*

Definition 16 (Dependency-Preserving).

A replicated execution xs *of a context* $\mathbf{c}$ *is dependency-preserving iff for every pair of requests* r *and* r' *in* R*, such that* $\mathit{call}_{\mathbf{c}}(r') \not\perp \mathit{call}_{\mathbf{c}}(r)$*, if* $r \prec_{xs(\mathit{orig}_{\mathbf{c}}(r'))} r'$*, then for every replica* n*,* $r \prec_{xs(n)} r'$*.*

Definition 17 (Weight). *weight is map from request identifiers* r *to the the maximum change* $\mathit{call}(r)$ *can have on any state* σ*.*

Formally, $\mathit{for\ all} r, \mathit{call}, \sigma, \mathsf{update}(\mathit{call}(r))(\sigma) = \sigma', \mathit{weight}(r) = \max(\Delta(\sigma, \sigma'))$

Lemma 6. *For every* h*,* t*,* xs*,* n*,* n'*, if* $w_0 \longrightarrow^* \langle h, t, xs, \_, \_\rangle$ *where* $h(n) = \langle \_, \_, r_b\rangle$*,* $h(n') = \langle \_, \_, r'_b\rangle$ *then* $xs(n) \setminus \{r_b\} \cup \{r_t \mid \langle n, r_t\rangle \in t\} = xs(n') \setminus \{r'_b\} \cup \{r'_t \mid \langle n', r'_t\rangle \in t\}$*.*

Proof.

In this lemma, we want to prove that at any time, for any two replicas, if the local buffers are excluded, they have the same set of requests identifiers either in the trace xs or in transmission t.

The proof is by induction and case analysis on $\longrightarrow$.
Initial state: In w_0, by definition $\mathsf{xs} = \emptyset$, $t = \emptyset$ and $h(n) = \langle \_, \_, \mathrm{id} \rangle$, $h(n') = \langle \_, \_, \mathrm{id} \rangle$. $\emptyset = \emptyset$ holds.
Let's assume the induction hypothesis:
for all $n, n', h_1, t_1, \mathsf{xs}_1$, if $w_0 \longrightarrow^* \langle h_1, t_1, \mathsf{xs}_1, \_, \_ \rangle$ where $h_1(n) = \langle \_, \_, r_{b1} \rangle$, $h_1(n') = \langle \_, \_, r'_{b1} \rangle$, then $\mathsf{xs}_1(n) \setminus \{r_{b1}\} \cup \{r_{t1} \mid \langle n, r_{t1} \rangle \in t_1\} = \mathsf{xs}_1(n') \setminus \{r'_{b1}\} \cup \{r'_{t1} \mid \langle n', r'_{t1} \rangle \in t_1\}$.
We want to prove that by take one more step, the equation still holds.
Case rule CALL:

It adds the call to xs for the current replica and to t for other replicas.
By taking this Call step:
$\mathsf{xs} = \mathsf{xs}_1[n \mapsto \mathsf{xs}_1 ::: r]$,
$t = t_1 \cup \{\langle n', r \rangle | n' \in \mathcal{N} \setminus \{n\}\}$
and the buffers of all the replicas stay the same $r_b = r_{b1}, r'_b = r'_{b1}$.
So on the left side of the equation, for the replica n which takes the Call step, we have:
$\mathsf{xs}(n) \setminus \{r_b\} \cup \{r_t \mid \langle n, r_t \rangle \in t\}$
$= \mathsf{xs}_1(n)[n \mapsto \mathsf{xs}_1 ::: r] \setminus \{r_{b1}\} \cup \{r_{t1} \mid \langle n, r_{t1} \rangle \in t_1\}$
$= \mathsf{xs}_1(n) \setminus \{r_{b1}\} \cup \{r_{t1} \mid \langle n, r_{t1} \rangle \in t_1\} \cup \{r\}$
On the right side of the equation, for all the other replicas which do not take the Call step, we have:
$\mathsf{xs}(n') \setminus \{r'_b\} \cup \{r'_t \mid \langle n', r'_t \rangle \in t\}$
$= \mathsf{xs}_1(n') \setminus \{r'_{b1}\} \cup \{r'_{t1} \mid \langle n', r'_{t1} \rangle \in t_1 \cup \{\langle n', r \rangle | n' \in \mathcal{N} \setminus \{n\}\}\}$
$= \mathsf{xs}_1(n') \setminus \{r'_{b1}\} \cup \{r'_{t1} \mid \langle n', r'_{t1} \rangle \in t_1\} \cup \{r\}$
Based on the induction hypothesis, we know that the left side of the equation is equal to the right side if we take Call step.

Case rule DELIVER:

It removes a request from t and adds it to xs.
By taking the Deliver step:
$\mathsf{xs} = \mathsf{xs}_1[n \mapsto \mathsf{xs}_1 ::: r]$,
$t = t_1 \setminus \{\langle n, r \rangle\}$,
and the buffers of all the replicas stay the same $r_b = r_{b1}, r'_b = r'_{b1}$.
So on the left side of the equation, for the replica n which takes the Deliver step, we have:
$\mathsf{xs}(n) \setminus \{r_b\} \cup \{r_t \mid \langle n, r_t \rangle \in t\}$
$= \mathsf{xs}_1(n)[n \mapsto \mathsf{xs}_1 ::: r] \setminus \{r_{b1}\} \cup \{r_{t1} \mid \langle n, r_{t1} \rangle \in t_1 \setminus \{\langle n, r \rangle\}\}|$
$= \mathsf{xs}_1(n) \setminus \{r_{b1}\} \cup \{r_{t1} \mid \langle n, r_{t1} \rangle \in t_1\}$
On the right side of the equation, for all the other replicas which do not take the Deliver step, their states stay unchanged:
$\mathsf{xs}(n') \setminus \{r'_b\} \cup \{r'_t \mid \langle n', r'_t \rangle \in t\} = \mathsf{xs}_1(n') \setminus \{r'_{b1}\} \cup \{r'_{t1} \mid \langle n', r'_{t1} \rangle \in t_1\}$
The two sides of the equation stay equal if we take the Deliver step.

Case rule CALLLOCAL:

It may or may not add the buffer identifier into xs based on different situations. But in the end, the buffer identifier is excluded from the equation on both sides.

By taking this CallLocal step,
$\mathsf{xs} = \begin{cases} \mathsf{xs}_1[n \mapsto \mathsf{xs}_1(n) ::: r] & \text{if } \mathsf{call}_1(r) = \mathsf{id} \\ \mathsf{xs}_1 & \text{else} \end{cases}$
$t = t_1$, $r'_b = r'_{b1}$ and the buffer identifier for replica n stays unchanged as r.
So on the left side of the equation, for the replica n which takes the CallLocal step, the state stays the same. This is because by contrapositive of Lemma 9 and $\mathsf{call}_1(r) = \text{id}$, we know that $r \notin \mathsf{xs}_1(n)$.
$\mathsf{xs}(n) \setminus \{r\} \cup \{r_t \mid \langle n, r_t\rangle \in t\}$
$= \mathsf{xs}_1(n)[n \mapsto \mathsf{xs}_1(n) ::: r] \setminus \{r\} \cup \{r_{t1} \mid \langle n, r_{t1}\rangle \in t_1\}$
or $\mathsf{xs}_1(n) \setminus \{r\} \cup \{r_{t1} \mid \langle n, r_{t1}\rangle \in t_1\}$
$= \mathsf{xs}_1(n) \setminus \{r\} \cup \{r_{t1} \mid \langle n, r_{t1}\rangle \in t_1\}$
On the right side of the equation, for all the other replicas which do not take the CallLocal step, their states stay unchanged:
$\mathsf{xs}_1(n') \setminus \{r'_{b1}\} \cup \{r'_{t1} \mid \langle n', r'_{t1}\rangle \in t_1\}$
The two sides of the equation stay equal if we take Deliver step.

Case rule SendBuffer:

For the replica which takes this SendBuffer step, the left side of equation contains one more element r. This is because according to Lemma 8, $r_{b1} \in \mathsf{xs}(n)$. Further, previously r_{b1} was excluded. Now with the a fresh buffer identifier, which has $\mathsf{call}(r_b) = \text{id}$ and has not been added to xs, the left side to the equation subtracts one less r. For all the other replicas, the right side of the equation also contain one one more element due to the new packages in transmit.
By taking this SendBuffer step, xs of all replicas stay unchanged. Notice that SendBuffer is the only rule that changes the buffer identifier with a fresh r:
$\mathsf{xs} = \mathsf{xs}_1$,
$t = t_1 \cup \{\langle n', r^*\rangle | n' \in \mathcal{N} \setminus \{n\}\}$, r_b is fresh and $r'_b = r'_{b1}$.
So on the left side of the equation, for the replica n which takes the SendBuffer step, the set contains one more element r:
$\mathsf{xs}(n) \setminus \{r_b\} \cup \{r_t \mid \langle n, r_t\rangle \in t\}$
$= \mathsf{xs}_1(n) \setminus \{r_b\} \cup \{r_{t1} \mid \langle n, r_{t1}\rangle \in t_1\} \cup \{r\}$
On the right side of the equation, for all the other replicas which do not take the SendBuffer step, their sets contain one more element r:
$\mathsf{xs}_1(n') \setminus \{r'_{b1}\} \cup \{r'_{t1} \mid \langle n', r'_{t1}\rangle \in t_1 \cup \{\langle n', r^*\rangle | n' \in \mathcal{N} \setminus \{n\}\}$
$= \mathsf{xs}_1(n') \setminus \{r'_{b1}\} \cup \{r'_{t1} \mid \langle n', r'_{t1}\rangle \in t_1\} \cup \{r\}$
The two sides of the equation stay equal if we take SendBuffer step.

Case rule DeliverBuffer:

The state transition is the same as Deliver rule.

We can conclude that the lemma holds.

Lemma 7. *For every h, xs, call, n, r_b, if $w_0 \longrightarrow^* \langle h, \_, \mathsf{xs}, \_, \mathsf{call}\rangle$, $h(n) = \langle \_, \_, r_b\rangle, \mathsf{call}(r_b) \neq id$ then $r_b \in \mathsf{xs}(n)$*

This lemma states that all the buffer identifiers r_b in $h(n) = \langle \_, \_, r_b\rangle$ which are not id have been added to $\mathsf{xs}(n)$.

Proof.

By induction and case analysis on $\longrightarrow$.

Initial state: In w_0, by definition $\mathsf{for\ all} n, h(n) = \langle s_n, \sigma_0, \mathrm{id}\rangle$; Therefore the lemma holds trivially.

Let's assume the base case: $\mathsf{for\ all}\ n, h_1, \mathsf{xs}_1, \mathsf{call}_1, w_0 \longrightarrow^* \langle h_1, \_, \mathsf{xs}_1, \_, \mathsf{call}_1\rangle$ if $h_1(n) = \langle \_, \_, r'\rangle, \mathsf{call}(r') \neq \mathrm{id}$, then $r' \in \mathsf{xs}_1(n)$.

We want to prove that by taking one more step, the statement still holds.

We note that only CallLocal and SendBuffer rules make changes regarding to the buffer identifier. All the other rules do not change any variables about the buffer, including call, orig, xs and r'.

Case rule SENDBUFFER:

It initializes a new buffer identifier r which has $\mathsf{call}(r) = \mathrm{id}$.

This initialization does not satisfy the precondition. So after SendBuffer step, the lemma still holds due to induction hypothesis.

Case rule CALLLOCAL:

It adds buffer identifier r to the xs according to the following condition and changes call. By taking this CallLocal step:

$\mathsf{call}(r) = c \cdot \mathsf{call}_1(r)$,

$$\mathsf{xs} = \begin{cases} \mathsf{xs}_1[n \mapsto \mathsf{xs}_1(n) ::: r] & \text{if } \mathsf{call}_1(r) = \mathsf{id} \\ \mathsf{xs}_1 & \text{else} \end{cases}$$

We need to consider two conditions. First condition, if $\mathsf{call}_1(r) = \mathrm{id}$, after the CallLocal step, $\mathsf{call}(r) \neq \mathrm{id}$ and $r \in \mathsf{xs}(n)$; thus, the lemma still holds.

Second condition, if $\mathsf{call}_1(r) \neq \mathrm{id}$, based on the induction hypothesis, $r \in \mathsf{xs}_1(n)$; thus the lemma still holds.

Lemma 8. *For every t, xs, orig, call, if $w_0 \longrightarrow^* \langle \_, t, \mathsf{xs}, \mathsf{orig}, \mathsf{call}\rangle$, then for all $\langle n, r\rangle \in t$, $r \in \mathsf{xs}(\mathsf{orig}(r))$*

In this lemma, we prove that all the request identifiers contained in the transmit package are executed in its orig's xs.

Proof.

By induction and case analysis on $\longrightarrow$.

Initial state: In w_0, by definition $t = \emptyset$; thus, the lemma holds trivially.

The induction hypothesis is as follow:

For every $t_1, \mathsf{xs}_1, \mathsf{orig}_1, \mathsf{call}_1$, if $w_0 \longrightarrow^* \langle \_, t_1, \mathsf{xs}_1, \mathsf{orig}_1, \mathsf{call}_1\rangle$ then $\mathsf{for\ all}\langle n, r\rangle \in t_1$, $r \in \mathsf{xs}_1(\mathsf{orig}_1(r))$.

We want to prove that by taking one more step, the statement still holds.

First of all, only Call and SendBuffer rules add new packages in transmit t.

Case rule SENDBUFFER:

It adds the call to t for other replicas and does not change xs.

By taking this SendBuffer step:

$t = t_1 \cup \{\langle n', r^*\rangle | n' \in \mathcal{N} \setminus \{n\}\}$, $\mathsf{orig} = \mathsf{orig}_1[r' \mapsto n]$, $\mathsf{call} = \mathsf{call}_1[r' \mapsto \mathrm{id}]$

Notice that r is the old buffer identifier in the SendBuffer pre-state.

By *Lemma* 7, we have that if $\mathsf{call}_1(r) \neq \mathrm{id}$, then $r \in \mathsf{xs}_1(n)$.

The lemma holds by the fact that $\mathsf{orig}(r) = n$ and $r \in \mathsf{xs}_1(n) = \mathsf{xs}(n)$.

Case rule CALL:

It adds the call to xs for the current replica and to t for other replicas.
By taking this Call step:
$\mathsf{xs} = \mathsf{xs}_1[n \mapsto \mathsf{xs}_1 ::: r]$, $\mathsf{orig} = \mathsf{orig}_1[r \mapsto n]$
$t = t_1 \cup \{\langle n', r\rangle | n' \in \mathcal{N} \setminus \{n\}\}$
The statement holds. Because for the request identifier r in the newly added packages to t, $r \in \mathsf{xs}(n)$ and $\mathsf{orig}(r) = n$.

Lemma 9. *For every xs, call and r, if* $w_0 \longrightarrow^* \langle \_, \_, \mathit{xs}, \_, \mathit{call}\rangle$ *and* $r \in \mathit{xs}$ *then* $\mathit{call}(r) \neq \mathit{id}$

We prove that xs only contains request that $\mathsf{call}(r) \neq \mathrm{id}$

Proof.

By induction and case analysis on $\longrightarrow$.
Initial state: In w_0, by definition $\mathsf{xs} = \emptyset$; thus, the lemma holds trivially.
The induction hypothesis is as below:
$\mathsf{for\ all}\ \mathsf{xs}_1, \mathsf{call}_1, w_0 \longrightarrow^* \langle \_, \_, \mathsf{xs}_1, \_, \mathsf{call}_1\rangle$ if $\mathsf{for\ all}\ r \in \mathsf{xs}_1$, then $\mathsf{call}_1(r) \neq \mathsf{id}$.
We want to prove that by taking one more step, the statement still holds.
Case rule Call:
It is assumed that programs do not have id calls; thus, the lemma holds by induction hypothesis and $\mathsf{call}(r) \neq \mathrm{id}$.
Case rule CallLocal:
It adds the buffer identifier r to $\mathsf{xs}(n)$ when $\mathsf{call}_1(r) = \mathrm{id}$.
By taking this CallLocal step, $\mathsf{call} = \mathsf{call}_1[r \mapsto c \cdot \mathsf{call}_1(r)]$
Notice that $\mathsf{call}(r) = c \neq \mathrm{id}$. For the other request identifiers, the lemma holds by the induction hypothesis.
Case rule DeliverBuffer:
It add the request r^* from t to $\mathsf{xs}(n)$ for the current replica: $\mathsf{xs} = \mathsf{xs}_1(n \mapsto \mathsf{xs}_1(n) ::: r)$
From Lemma 8 we can know that $r \in \mathsf{xs}_1(\mathsf{orig}_1(r))$. By the induction hypothesis,we know that $\mathsf{call}_1(r) \neq \mathrm{id}$.
Case rule Deliver:
Similar to DeliverBuffer Case.
In conclusion, the lemma holds.

Lemma 10. *For every h, xs, n, n′, r and r′, if* $w_0 \longrightarrow^* \langle h, \emptyset, \mathit{xs}, \_, \_\rangle$ *where* $h(n) = \langle \_, \_, r\rangle$, $h(n') = \langle \_, \_, r'\rangle$ *and* $\mathit{call}(r) = \mathit{call}(r') = \mathit{id}$ *then* $\{\mathit{xs}(n)\} = \{\mathit{xs}(n')\}$.

In this lemma, we prove that when all the buffers are flushed and the transmitted messages are delivered, the set of requests of xs of any two replicas are the same. In another word, all the replicas have the same set of R in the end.

Proof.

Immediate from Lemma 6 and Lemma 9. When $t = \emptyset$ and $\mathsf{call}(r) = \mathsf{call}(r') =$

id, we have $\{\mathsf{xs}(n)\} = \{\mathsf{xs}(n')\}$.

Lemma 11. *For every h, t, call, n and r, if $w_0 \longrightarrow^* \langle h, t, \_, \_, \mathit{call}\rangle$, then for all $\langle n, r^*\rangle \in t$ or $h(n) = \langle \_, \_, r\rangle, \mathit{call}(r) \neq id$, we have $\mathsf{AllSComm}(\mathit{call}(r))$, $\mathsf{InvSuff}(\mathit{call}(r))$ and $\mathsf{LetPRComm}(\mathit{call}(r))$.*

We prove that all the buffers r satisfy AllSComm, InvSuff and LetPRComm conditions, no matter they are in transmit t or not.

Proof.

By induction and case analysis on $\longrightarrow$.

Initial state: In w_0, by definition $t = \emptyset$ and for all $n, h(n) = \langle \_, \_, \mathrm{id}\rangle$; thus, the lemma holds trivially.

The induction hypothesis is:

for all $h_1, t_1, \mathsf{call}_1, n$, if $w_0 \longrightarrow^* \langle h_1, t_1, \_, \_, \mathsf{call}_1\rangle$ then for all $\langle n, r^*\rangle \in t_1$, or $h_1(n) = \langle \_, \_, r\rangle, \mathsf{call}_1(r) \neq \mathrm{id}$, we have $\mathsf{AllSComm}(\mathsf{call}_1(r))$, $\mathsf{InvSuff}(\mathsf{call}_1(r))$ and $\mathsf{LetPRComm}(\mathsf{call}_1(r))$.

We need to proof that by taking one more step the statement still holds.

First we notice that only SendBuffer rule adds packages containing buffers in t. And only CallLocal rule changes the content of $\mathsf{call}(r)$, where r is a buffer identifier.

Case rule CALLLOCAL:

By taking this CallLocal step:

$\mathsf{call} = \mathsf{call}_1[r \mapsto c \cdot \mathsf{call}_1(r)]$

By the precondition in this rule, we have $\mathsf{InvSuff}(\mathsf{call}(r)), \mathsf{LetPRComm}(\mathsf{call}(r))$ and $\mathsf{AllSComm}(c)$. By the induction hypothesis we have $\mathsf{AllSComm}(\mathsf{call}_1(r))$.

So we have $\mathsf{AllSComm}(c \cdot \mathsf{call}_1(r)) = \mathsf{AllSComm}(\mathsf{call}(r))$. The lemma holds.

Case rule SENDBUFFER:

By taking this SendBuffer step:

$t = t_1 \cup \{\langle n', r^*\rangle | n' \in \mathcal{N} \setminus \{n\}\}, \mathsf{call} = \mathsf{call}_1[r' \mapsto \mathrm{id}]$

By the induction hypothesis and $\mathsf{call}(r) \neq \mathrm{id}$ in the precondition, we have $\mathsf{AllSComm}(\mathsf{call}_1(r))$, $\mathsf{InvSuff}(\mathsf{call}_1(r))$ and $\mathsf{LetPRComm}(\mathsf{call}_1(r))$. By the transition above we have $\mathsf{call}(r) = \mathsf{call}_1(r)$. In conclusion, the lemma holds.

Lemma 12. *If for all $n, n' \in \mathcal{N}$, for all $\mathit{xs}, \mathit{orig}, \mathit{call}$, if $w_0 \longrightarrow^* \langle \_, \_, \mathit{xs}, \mathit{orig}, \mathit{call}\rangle$, then xs is conflict-synchronizing in the context $\langle \mathit{orig}, \mathit{call}\rangle$.*

Proof.

By induction and case analysis on $\longrightarrow$.

Initial state: In w_0, by definition $\mathsf{xs} = \emptyset$, the lemma holds trivially.

The induction hypothesis is:

$w_0 \longrightarrow^* \langle \_, \_, \mathsf{xs}_1, \mathsf{orig}_1, \mathsf{call}_1\rangle$. for all $r, r', \mathsf{call}_1(r) \bowtie \mathsf{call}_1(r')$

- if $r \in \mathsf{xs}_1(n)$ and $r' \in \mathsf{xs}_1(n')$ then $r' \in \mathsf{xs}_1(n)$ or $r \in \mathsf{xs}_1(n')$.
- if $r' \prec_{\mathsf{xs}_1(n')} r$, $r \in \mathsf{xs}_1(n)$, then $r' \in \mathsf{xs}_1(n)$.
- if $r' \prec_{\mathsf{xs}_1(n')} r$ and $r, r' \in \mathsf{xs}_1(n)$, then $r' \prec_{\mathsf{xs}_1(n)} r$.

We need to prove that by taking one more step the statement still holds.

Case rule CALL:

By taking this Call step:
$\mathsf{call} = \mathsf{call}_1[r \mapsto c]$, $\mathsf{orig} = \mathsf{orig}_1[r \mapsto n]$, $\mathsf{xs} = \mathsf{xs}_1[n \mapsto (\mathsf{xs}_1(n) ::: r)]$
By ConflictSyncInit condition, for all $n', r'.r' \in \mathsf{xs}(n') \wedge \mathsf{call}(r) \bowtie \mathsf{call}(r')$ we have $r' \in \mathsf{xs}(n)$. Further we have $r \in \mathsf{xs}(n)$. This proves condition 1.
Because the request identifier r is fresh, $r \notin \mathsf{xs}(n')$ for all $n' \in \mathcal{N} \setminus n$. This proves condition 2 and 3.

Case rule DELIVER:

By taking this Deliver step:
$\mathsf{call} = \mathsf{call}_1$, $\mathsf{orig} = \mathsf{orig}_1$, $\mathsf{xs} = \mathsf{xs}_1[n \mapsto (\mathsf{xs}_1(n) ::: r)]$
Notice that by Lemma 8, $r \in \mathsf{xs}_1(\mathsf{orig}_1(r))$. So there exists $n'' \neq n$, by the induction hypothesis, we have $r \in \mathsf{xs}_1(n'') = \mathsf{xs}(n'')$ and $r' \in \mathsf{xs}_1(n') = \mathsf{xs}(n')$ then $r' \in \mathsf{xs}_1(n'') = \mathsf{xs}(n'')$ or $r \in \mathsf{xs}_1(n') = \mathsf{xs}(n')$.
We prove the first condition by contradiction. The contradiction assumption is:
There exists $r \in \mathsf{xs}(n), r' \in \mathsf{xs}(n')$ and $r' \notin \mathsf{xs}(n) \wedge r \notin \mathsf{xs}(n')$
By the contradiction assumption and induction hypothesis, we have $r' \in \mathsf{xs}(n'')$. There are two possible orders between r and r' at replica n'':
If $r' \prec_{\mathsf{xs}(n'')} r$, by ConflictSync conditions $r' \prec_{\mathsf{xs}(n)} r$, which contradicts the assumption $r' \notin \mathsf{xs}(n)$.
If $r \prec_{\mathsf{xs}(n'')} r'$, by contradiction assumption we have $r \notin \mathsf{xs}_1(n') = \mathsf{xs}(n')$, $r' \in \mathsf{xs}_1(n') = \mathsf{xs}(n')$.
By induction hypothesis on condition 2, $r \in \mathsf{xs}_1(n') = \mathsf{xs}(n')$. We reach a contradiction.
In conclusion, the first condition holds by taking Deliver step.
The other two conditions holds trivially by ConflictSync condition in Deliver step and the fact that r is the last request identifier in $\mathsf{xs}(n)$.

Case rule CALLLOCAL:

After taking the CallLocal step:
$$\mathsf{xs} = \begin{cases} \mathsf{xs}_1[n \mapsto (\mathsf{xs}_1(n) ::: r)] & \text{if } \mathsf{call}_1(r) = \mathsf{id} \\ \mathsf{xs}_1 & \text{else} \end{cases}$$
$\mathsf{call} = \mathsf{call}_1[r \mapsto c']$
The only tentative newly added request identifier is the buffer identifier r.
By the precondition $\mathsf{AllSComm}(c)$, $\mathsf{InvSuff}(c')$, $\mathsf{LetPRComm}(c')$ and Lemma 11,

we have that for all r', r' concurs with r.

Case rule SENDBUFFER:

Trivial. After taking SendBuffer step, $\mathsf{xs} = \mathsf{xs}_1$. The lemma holds because of the induction hypothesis.

Case rule DELIVERBUFFER:

The only newly added identifier in xs is a buffer identifier, which by Lemma 11, similar to the CallLocal step, concurs with all the other $r' \in \mathsf{xs}$.
The lemma holds trivially by taking DeliverBuffer step.

We can conclude that the lemma holds by the induction above.

Lemma 13. *for all* $n, n' \in \mathcal{N}$, *for all* $r, r' \in R, h, \mathsf{xs}, \mathsf{orig}, \mathsf{call}$, *if* $w_0 \longrightarrow^* \langle h, \emptyset, \mathsf{xs}, \mathsf{orig}, \mathsf{call}\rangle$, $h(n) = \langle \_, \_, r_b\rangle, h(n') = \langle \_, \_, r'_b\rangle, \mathsf{call}(r_b) = \mathsf{call}(r'_b) = id, \mathsf{call}(r) \bowtie \mathsf{call}(r')$, *in the context* $\langle \mathsf{orig}, \mathsf{call}\rangle$, $r' \prec_{\mathsf{xs}(n')} r$ *then* $r' \prec_{\mathsf{xs}(n)} r$

We want to prove that in a complete execution for n and n', the trace xs have total order for all the conflicting request identifiers.
By Lemma 10 and Def. 4, we have a complete replicated execution for n and n'. By Lemma 12 and Def. 13, we have a total order of all the conflicting request identifiers on the complete replicated execution.

Lemma 14. *for all* h, xs, n, *and* σ, *if* $w_0 \longrightarrow^* \langle h, \_, \mathsf{xs}, \_, \_\rangle$, *where* $h(n) = \langle \_, \sigma, \_\rangle$ *and* $\mathbf{ss}$ *is the state function of* xs *then* $\sigma = \mathbf{ss}(n)(|\mathsf{xs}(n)|)$.

Proof.
By induction and case analysis on $\longrightarrow$.
Initial state: In w_0, by definition $\mathsf{for\ all}\ n, \sigma_0 = \mathbf{ss}(n)(|\emptyset|)$; thus, the lemma holds trivially.
The induction hypothesis is:
$\mathsf{for\ all}\ n, h_1, \mathsf{xs}_1, w_0 \longrightarrow^* \langle \mathsf{h}_1, \_, \mathsf{xs}_1, \_, \_\rangle, h_1(n) = \langle \_, \sigma_1, \_\rangle$ and $\sigma_1 = \mathbf{ss}(n)(|\mathsf{xs}_1(n)|)$
We need to proof that by taking one more step the statement still holds.
Case rule CALL:
By taking this Call step:
$h[n \mapsto (\_, \sigma, \_)]$, $c(\sigma_1) = \langle \_, \sigma, \_\rangle$, $\mathsf{xs} = \mathsf{xs}_1[n \mapsto (\mathsf{xs}_1(n) ::: r)]$, $\mathsf{call} = \mathsf{call}_1[r \mapsto c]$
By Def. 8, we have:
$\mathbf{ss}(n)(|\mathsf{xs}(n)|) = \mathbf{ss}(|\mathsf{xs}_1(n)| + 1) = \mathsf{update}(\mathsf{call}(r))(\mathbf{ss}(n)(|\mathsf{xs}_1(n)|)) = \mathsf{update}(c)(\sigma_1) = \sigma$
Case rule DELIVER:
By taking this Deliver step:
$h[n \mapsto (\_, \sigma, \_)]$, $\mathsf{call}(r)(\sigma_1) = \langle \_, \sigma, \_\rangle$, $\mathsf{xs} = \mathsf{xs}_1[n \mapsto (\mathsf{xs}_1(n) ::: r)]$,
By Def. 8, we have:
$\mathbf{ss}(n)(|\mathsf{xs}(n)|) = \mathbf{ss}(|\mathsf{xs}_1(n)| + 1) = \mathsf{update}(\mathsf{call}(r))(\mathbf{ss}(n)(|\mathsf{xs}_1(n)|)) = \mathsf{update}(\mathsf{call}(r))(\sigma_1) = \sigma$
Case rule CALLLOCAL:
By taking this CallLocal step:
$h[n \mapsto (\_, \sigma, \_)]$, $c(\sigma_1) = \langle \_, \sigma, \_\rangle$, $\mathsf{call} = \mathsf{call}_1[r \mapsto c \cdot \mathsf{call}_1(r)]$,
$\mathsf{xs}' = \begin{cases} \mathsf{xs}_1[n \mapsto (\mathsf{xs}_1(n) ::: r)] & \text{if } \mathsf{call}_1(r) = \mathsf{id} \\ \mathsf{xs}_1 & \text{else} \end{cases}$
By Def. 8, we have:
If $\mathsf{call}_1(r) = \mathrm{id}$, then $\mathbf{ss}(n)(|\mathsf{xs}(n)|) = \mathbf{ss}(|\mathsf{xs}_1(n)|+1) = \mathsf{update}(\mathsf{call}(r))(\mathbf{ss}(n)(|\mathsf{xs}_1(n)|))$

$= \mathsf{update}(c)(\sigma_1) = \sigma$
If $\mathsf{call}_1(r) \neq \mathrm{id}$, we need to consider the situation that $r \neq \mathsf{last}(\mathsf{xs}(n))$.
Let $i = \mathsf{xs}(n)^{-1}(r)$. All the request identifiers from $i + 1$ to $|\mathsf{xs}_1(n)|$ in $\mathsf{xs}_1(n)$ are not in the current buffer but state commute with the new call

c by $\mathsf{AllSComm}(c)$ condition. By Lemma 31, the two executions below have the same final state:
$\mathsf{x} = \mathsf{xs}_1(n)[i...|\mathsf{xs}_1(n)|] \cdot c$,
$\mathsf{x}' = \mathsf{xs}_1(n)[i...|\mathsf{xs}_1(n)|][i+1 \mapsto c][i+2 \mapsto \mathsf{xs}_1(n)(i+1)]... \cdot [|\mathsf{xs}_1(n)|+1 \mapsto \mathsf{xs}_1(n)(|\mathsf{xs}_1(n)|)]$
By the induction hypothesis:
$\sigma = \mathsf{update}(c)(\sigma_1) = \mathsf{update}(\mathsf{x})(\mathbf{ss}(n)(i)) = \mathsf{update}(\mathsf{x}')(\mathbf{ss}(n)(i)) = \mathsf{update}(\mathsf{xs}(n)(|\mathsf{xs}(n)|)(\mathbf{ss}(n)(|\mathsf{xs}(n)|))) = \mathbf{ss}(n)(|\mathsf{xs}(n)|)$

Case rule SendBuffer:
By taking this SendBuffer step, the state and xs of no replica changes. The lemma holds by the induction hypothesis.

Case rule DeliverBuffer:
The transition is the same as Deliver rule.

Lemma 15 (Convergence). *For all h, n, n', σ, σ', r and r', if $w_0 \longrightarrow^* \langle h, \emptyset, \_, \_, \_\rangle$ where $h(n) = \langle \_, \sigma, r\rangle$, $h(n') = \langle \_, \sigma', r'\rangle$ and $\mathsf{call}(r) = \mathsf{call}(r') = \mathsf{id}$ then $\sigma = \sigma'$.*

Proof.
Immediate from Lemma 13, Lemma 38 and Lemma 14.

Lemma 16. *For every xs, orig and call, if $w_0 \longrightarrow^* \langle \_, \_, \mathsf{xs}, \mathsf{orig}, \mathsf{call}\rangle$, then xs is locally-permissible in the context $\langle \mathsf{orig}, \mathsf{call}\rangle$.*

We want to prove that for every xs, orig and call, $w_0 \longrightarrow^* \langle \_, \_, \mathsf{xs}, \mathsf{orig}, \mathsf{call}\rangle$, in the context $\langle \mathsf{orig}, \mathsf{call}\rangle$ if $\mathsf{for\ all} r \in \mathsf{xs}, \mathcal{P}(\mathbf{s}(i), r)$,where x is $\mathsf{xs}(\mathsf{orig}(r))$, s is the state function of x and $i = x^{-1}(r)$.

Proof.
By induction and case analysis on $\longrightarrow$.
In the rule Call, locally executed calls are checked to be permissible.
In the rule CallLocal, locally executed calls are checked to be invariant sufficient, which means that they are permissible.

Lemma 17. *For every xs, orig and call, if $w_0 \longrightarrow^* \langle \_, \_, \mathsf{xs}, \mathsf{orig}, \mathsf{call}\rangle$, then xs is dependency-preserving in the context $\langle \mathsf{orig}, \mathsf{call}\rangle$.*

Proof.
By induction and case analysis on $\longrightarrow$.
Initial state: In w_0, by definition in P13, $\mathsf{xs} = \emptyset$, the lemma holds trivially.
The induction hypothesis is:
$\mathsf{for\ all}\ n, \mathsf{xs}_1, \mathsf{orig}_1, \mathsf{call}_1, w_0 \longrightarrow^* \langle \_, \_, \mathsf{xs}_1, \mathsf{orig}_1, \mathsf{call}_1\rangle$, such that $\mathsf{for\ all}\ r, r', \mathsf{call}_1(r) \not\perp \mathsf{call}_1(r')$ and $r' \prec_{\mathsf{xs}(\mathsf{orig}_1(r))} r$, if $r \in \mathsf{xs}_1(n)$ then $r' \in \mathsf{xs}_1(n) \wedge r' \prec_{(\mathsf{xs}_1(n))} r$
We need to proof that by taking one more step the statement still holds.
Notice that Call, CallLocal and SendBuffer rules are either executing fresh requests or does not change xs. We only need to consider Deliver and Deliver-

Buffer rules.
Case rule DELIVER:
By taking this Deliver step:
$\mathsf{call} = \mathsf{call}_1, \mathsf{orig} = \mathsf{orig}_1, \mathsf{xs} = \mathsf{xs}_1[n \mapsto (\mathsf{xs}_1(n) ::: r)]$,
By DepPres condition and the fact that $\mathsf{last}(\mathsf{xs}(n)) = r$, the lemma holds trivially.
Case rule DELIVERBUFFER:
By Lemma 11 and $\mathsf{call} = \mathsf{call}_1$, we know that $\mathsf{InvSuff}(\mathsf{call}(r))$. By Def. 9, we know that $\mathsf{call}(r)$ is independent of any calls. The lemma holds trivially.

Lemma 18 (Integrity). *For all h, call, n, r and σ, if $w_0 \longrightarrow^* \langle h, \_, \mathsf{xs}_1, \_, \mathsf{call}_1\rangle \xrightarrow{n,r,c} \langle \_, \_, \mathsf{xs}, \_, \mathsf{call}\rangle$, where $h(n) = \langle \_, \sigma, \_\rangle$ then $\mathsf{integrity}(\sigma, c)$.*

Proof.
For all the rules except CallLocal rule, we prove the lemma immediately from Lemma 16, Lemma 12, Lemma 17, Lemma 40, Lemma 14 and the fact that $\mathsf{last}(\mathsf{xs}(n)) = r$
For CallLocal rule, the integrity is a bit tricky. Because CallLocal rule applies a new call c on state σ. But c's request identifier(which is the buffer identifier r) has the possibility of not being the last request identifier in $\mathsf{xs}(n)$ when $\mathsf{call}_1(r) \neq \mathrm{id}$.
In this case the assumption below in Lemma 40 does not meet our expectation of proving integrity for every single call in the execution:
let $i = \mathsf{xs}(n)^{-1}(r)$ we prove $\mathsf{guard}(c)(\mathbf{ss}(n)(i)) \wedge \mathcal{I}(\mathbf{ss}(n)(i))$
Instead, we prove:
$\mathsf{guard}(c)(\sigma) \wedge \mathcal{I}(\sigma)$ for the single call c.
By precondition $\mathcal{P}(\sigma, c)$ of CallLocal rule, we have $\mathsf{guard}(c)(\sigma)$. By Lemma 16, Lemma 12, Lemma 17, Lemma 14 and Lemma 39 on xs_1, we have $\mathcal{I}(\sigma)$.
Thus the lemma holds.

Lemma 19. *For all h, h', n, n', t, xs, xs' and orig, $w_0 \longrightarrow^* \langle h, t, \mathsf{xs}, \_, \_\rangle$ $(\longrightarrow \cup \xrightarrow{n',\_,\_})^*$ $\langle h', \_, \mathsf{xs}', \mathsf{orig}, \_\rangle$, $h(n') = \langle s, \_, \_\rangle, h'(n') = \langle s, \_, \_\rangle$, for all r, if $r \in \mathsf{xs}'(n') \setminus \mathsf{xs}(n')$, $\mathsf{orig}(r) = n$ then $\langle n', r\rangle \in t$ or $h(n) = \langle \_, \_, r\rangle$.*

Proof.
By induction and case analysis on $\longrightarrow$.
Since the statement of n' stays unchanged in the second transition $(\longrightarrow \cup \xrightarrow{n',\_,\_})^*$, only the rules DELIVER and DELIVERBUFFER can add to xs. They receive a packages from the network t or newly added packages during the second transition. With the limited transitions, only the rule SENDBUFFER can add buffer r where $h(\mathsf{orig}(r)) = \langle \_, \_, r\rangle$ to the network t.

Lemma 20. *For all h, h', n, n', t, xs, xs' and orig, $w_0 \longrightarrow^* \langle h, t, \mathsf{xs}, \_, \_\rangle$ $(\longrightarrow \cup \xrightarrow{n',\_,\_})^*$ $\langle h', \_, \mathsf{xs}', \mathsf{orig}, \_\rangle$, $h(n') = \langle s, \_, \_\rangle, h'(n') = \langle s, \_, \_\rangle$, for all r, if $r \in \mathsf{xs}'(n') \setminus \mathsf{xs}(n')$, $\mathsf{orig}(r) = n$ then $r \in \mathsf{xs}(n) \setminus \mathsf{xs}(n')$*

Proof.
From Lemma 19 and Lemma 8, Lemma 7, we immediately get $r \in \mathsf{xs}(n)$. Combined with the precondition $r \notin \mathsf{xs}(n')$, the lemma is trivially true.

Lemma 21. *For all t, orig if $w_0 \longrightarrow^* \langle \_, t, \_, \mathit{orig}, \_\rangle$, then for all $\langle n, r\rangle \in t, \mathsf{orig}(r) \neq n$.*

In this lemma we want to prove that all request identifiers in transmit do not have the destination of $\mathsf{orig}(r)$.

Proof.
Immediate from induction and case analysis on $\longrightarrow$.
Initial state: In w_0, by definition $\mathsf{xs} = \emptyset$, the lemma holds trivially.
The induction hypothesis is:
for all $t_1, \mathsf{orig}_1, w_0 \longrightarrow^* \langle \_, t_1, \_, \mathsf{orig}_1, \_\rangle$. such that for all $\langle n, r\rangle \in t_1, \mathsf{orig}_1 \neq n$
Case rule Call, SendBuffer:
By taking these two steps:
$t = t_1 \cup \{\langle n', r\rangle | n' \in \mathcal{N} \setminus \{n\}\}$, $\mathsf{orig}(r) = n$
For all the newly added r in t, $n' \neq \mathsf{orig}(r)$. The lemma still holds.
Case rule CallLocal, Deliver, DeliverBuffer:
In these rules t does not change or have less packages in it. $\mathsf{orig} = \mathsf{orig}_1$.
By induction hypothesis, the lemma holds trivially.

Lemma 22. *For all xs, orig, call and n, if $w_0 \longrightarrow^* \langle \_, \_, \mathit{xs}, \mathit{orig}, \mathit{call}\rangle$, then $\mathsf{InBound}_{\langle \mathit{orig}, \mathit{call}\rangle}(\mathit{xs}, n)$.*

In this lemma we want to prove that at any point of time, all the replicas in the world maintains InBound property.

Proof.
Immediate from induction and case analysis on $\longrightarrow$.
Initial state: In w_0, by definition $\mathsf{xs} = \emptyset$, the lemma holds trivially.
Let's assume the base case:
for all $n, \mathsf{xs}_1, \mathsf{orig}_1, \mathsf{call}_1, w_0 \longrightarrow^* \langle \_, \_, \mathsf{xs}_1, \mathsf{orig}_1, \mathsf{call}_1\rangle$. such that $\mathsf{InBound}_{\langle \mathsf{orig}_1, \mathsf{call}_1\rangle}(\mathsf{xs}_1, n)$
Case rule Call, CallLocal:
By InBound in the precondition for these two rules, the lemma holds trivially.
Case rule SendBuffer:
$\mathsf{xs} = \mathsf{xs}_1$, for all $r \in \mathsf{xs}(n), \mathsf{call}(r) = \mathsf{call}_1(r) \wedge \mathsf{orig}(r) = \mathsf{orig}_1(r)$. By induction hypothesis the lemma holds trivially.
Case rule Deliver, DeliverBuffer:
By taking these two steps: $\mathsf{orig} = \mathsf{orig}_1, \mathsf{call} = \mathsf{call}_1$. By Lemma 21 and induction hypothesis, the lemma holds trivially.

Lemma 23. *For all h, h', n, n', t, xs, xs' and orig, if $w_0 \longrightarrow^* \langle h, t, \mathit{xs}, \_, \_\rangle$ $(\longrightarrow \cup \xrightarrow{n', \_, \_})^* \langle h', \_, \mathit{xs}', \mathit{orig}, \_\rangle$, $h(n') = \langle s, \_, \_\rangle, h'(n') = \langle s, \_, \_\rangle$, then for all r, $\sum_{r \in \mathit{xs}'(n') \setminus \mathit{xs}(n')} \mathit{weight}(\mathit{call}(r)) < \epsilon$*

In this lemma we want to prove at any time, for all the replicas the sum of weight of unknown calls are bounded.

Proof.

From Lemma 20 and Lemma 22, we have

$\sum_{r \in \mathsf{xs}'(n')\setminus\mathsf{xs}(n')} \mathsf{weight}(\mathsf{call}(r)) = \sum_{n\in\mathcal{N}\setminus\{n'\}} \sum_{r \in \mathsf{xs}'(n')\setminus\mathsf{xs}(n')\wedge\mathsf{orig}(r)=n} \mathsf{weight}(\mathsf{call}(r))$

$< \frac{\epsilon}{|\mathcal{N}|-1} \times (N-1) < \epsilon$

Lemma 24 (Recency). *For all h, h', n, s, σ and σ', if $w_0 \longrightarrow^* \langle h, \_, \_, \_, \_\rangle$ $(\longrightarrow \cup \xrightarrow{n,\_,\_})^* \langle h', \_, \_, \_, \_\rangle$, $h(n) = \langle s, \sigma, \_\rangle$, and $h'(n) = \langle s, \sigma', \_\rangle$ then $\Delta(\sigma', \sigma) < \epsilon$.*

Proof.

Immediate from Lemma 23 and Lemma 14.

Lemma 25. *A replicated execution xs is conflict-synchronizing iff it is $\mathcal{S}$-conflict-synchronizing and $\mathcal{P}$-conflict-synchronizing.*

Proof. Immediate from Def. 13, Def. 14, and Def. 15.

Lemma 26. *For every execution x, the precedence order $\prec_{\mathsf{x}}$ is transitive.*

Proof. Immediate from Def. 2.

Lemma 27. *In a context c, for every pair of executions x and x' with state functions $\mathbf{s}$ and $\mathbf{s}'$ and every position i, if $\mathbf{s}'(i) = \mathbf{s}(i)$ and $\mathsf{x}'[i..j] = \mathsf{x}[i..j]$ then for every $i \leq k \leq j+1$, $\mathbf{s}'(k) = \mathbf{s}(k)$*

Proof.
Simple induction on i and Def. 2.

Lemma 28. *In every $\mathcal{S}$-conflict-synchronizing replicated execution xs, for every request r, r' and r'', if r precede r' in the execution of a node, and r' precedes r'' but r does not precede r'' in the execution of another node, then r and r' $\mathcal{S}$-commute.*

Formally, for every context $\mathbf{c}$ and $\mathcal{S}$-conflict-synchronizing replicated execution xs, for every requests r, r', r'' and nodes n and n', if

- $r \prec_{\mathsf{xs}(n)} r'$
- $r' \prec_{\mathsf{xs}(n')} r''$
- $r \not\prec_{\mathsf{xs}(n')} r''$

then $\mathsf{call}_{\mathbf{c}}(r) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(r')$.

Proof.
We assume that

(1) xs, a replicated execution
(2) xs is $\mathcal{S}$-conflict-synchronizing
(3) $r \prec_{\mathsf{xs}(n)} r'$
(4) $r' \prec_{\mathsf{xs}(n')} r''$
(5) $r \not\prec_{\mathsf{xs}(n')} r''$

We show that

$\mathsf{call}_{\mathbf{c}}(r) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(r')$.

From [1], we have that
(6) $\prec_{\mathsf{xs}(n')}$ is a total order.
From [6] and [5],
(7) $r'' \prec_{\mathsf{xs}(n')} r$
By Lemma 26 on [4] and [7], we have
(8) $r' \prec_{\mathsf{xs}(n')} r$
From Def. 14 on [2] and [3], and [8], we have
$\mathsf{call}_{\mathbf{c}}(r) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(r')$

Lemma 29. *In every $\mathcal{P}$-conflict-synchronizing replicated execution xs, for every request r, and r' such that either $\mathsf{call}_{\mathbf{c}}(r)$ or $\mathsf{call}_{\mathbf{c}}(r')$ is not invariant-sufficient, if r precede r' in the execution of a node, but does not precede it in the execution of another node, then r $\mathcal{P}$-R-commutes with r'.*

Formally, for every context $\mathbf{c}$ and $\mathcal{P}$-conflict-synchronizing replicated execution xs, for every requests r, and r' such that either $\mathsf{call}_{\mathbf{c}}(r)$ or $\mathsf{call}_{\mathbf{c}}(r')$ is not invariant-sufficient, and nodes n and n', if

- $r \prec_{\mathsf{xs}(n)} r'$
- $r' \not\prec_{\mathsf{xs}(n')} r$

then $\mathsf{call}_{\mathbf{c}}(r) \rightarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r')$.

Proof.
Similar to Lemma 28 using Def. 15.

Lemma 30. *In every dependency-preserving replicated execution xs, for every request r, and r', if $\mathsf{call}_{\mathbf{c}}(r')$ is not invariant-sufficient and r precede r' in the execution of the originating node of r', but does not precede it in the execution of another node, then r' $\mathcal{P}$-L-commutes r.*

Formally, for every context $\mathbf{c}$ and dependency-preserving replicated execution xs, for every request r and r' and node n, if $\mathsf{call}_{\mathbf{c}}(r')$ is not invariant-sufficient and

- $r \prec_{\mathsf{xs}(\mathsf{orig}_{\mathbf{c}}(r'))} r'$
- $r' \not\prec_{\mathsf{xs}(n)} r$

then $\mathsf{call}_{\mathbf{c}}(r') \leftarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r)$.

Proof.
Similar to Lemma 28 using Def. 16.

Lemma 31. *In every execution, if a request r at a position i $\mathcal{S}$-commutes with subsequent requests R up to and including a position j, then shifting left R by one position and moving r to j keeps the post-state of j the same.*

Formally, for every context $\mathbf{c}$ and execution x, for every position i and j such that $0 \leq i < j < |\mathcal{R}(\mathsf{x})|$, if

- *for every position k, $i < k \leq j$, $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(i)) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}(k))$,*
- *let x' be $\mathsf{x}[i \mapsto \mathsf{x}(i+1)]..[j-1 \mapsto \mathsf{x}(j)][j \mapsto \mathsf{x}(i)]$, and*
- *let $\mathbf{s}$ and $\mathbf{s}'$ be the state functions for x and x',*

then $\mathbf{s}'(j+1) = \mathbf{s}(j+1)$.

Proof.
We prove the following stronger statement.

For every context $\mathbf{c}$ and execution x, for every position i, j and l such that $0 \leq i \leq l \leq j < |\mathcal{R}(\mathsf{x})|$, if

- for every position k, $i < k \leq j$, $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(i)) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}(k))$,
- $\mathsf{x}' =$
  x if $l = i$
  $\mathsf{x}[i \mapsto \mathsf{x}(i+1)]..[l-1 \mapsto \mathsf{x}(l)][l \mapsto \mathsf{x}(i)]$ otherwise,
- $\mathbf{s}$ and $\mathbf{s}'$ are the state functions for x and x',

then $\mathbf{s}'(j+1) = \mathbf{s}(j+1)$.

The original statement is derived by setting l to j.

We assume that
(1) for every position k, $i < k \leq j$,
$\mathsf{call}_{\mathbf{c}}(\mathsf{x}(i)) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}(k))$.
(2) $\mathsf{x}' = \mathsf{x}[i \mapsto \mathsf{x}(i+1)]..[l-1 \mapsto \mathsf{x}(l)][l \mapsto \mathsf{x}(i)]$,
(3) $\mathbf{s}$ and $\mathbf{s}'$ are the state functions for x and x',
We prove that
$\mathbf{s}'(j+1) = \mathbf{s}(j+1)$.

Proof by induction on l.
Base case:
$l = i$
Trivial.
Inductive case:
We assume that
(4) $\mathbf{s}'(j+1) = \mathbf{s}(j+1)$
(5) $\mathsf{x}'' = \mathsf{x}[i \mapsto \mathsf{x}(i+1)]..[l \mapsto \mathsf{x}(l+1)][l+1 \mapsto \mathsf{x}(i)]$,
(6) $\mathbf{s}''$ is the state function for x'',
We prove that
$\mathbf{s}''(j+1) = \mathbf{s}(j+1)$.

From [2] and [5], we have
(7) $\mathsf{x}'' = \mathsf{x}'[l \mapsto \mathsf{x}(l+1)][l+1 \mapsto \mathsf{x}(i)]$,
From Def. 8 on [3], [2], we have
(8) $\mathbf{s}'(l+2) = \mathsf{update}(\mathsf{call}_c(\mathsf{x}(l+1)))$
$(\mathsf{update}(\mathsf{call}_c(\mathsf{x}(i)))(\mathbf{s}'(l)))$
From Def. 8 on [3], [7], we have
(9) $\mathbf{s}''(l+2) = \mathsf{update}(\mathsf{call}_c(\mathsf{x}(i)))$
$(\mathsf{update}(\mathsf{call}_c(\mathsf{x}(l+1)))(\mathbf{s}'(l)))$
From [1], Def. 1, [8] and [9], we have
(10) $\mathbf{s}''(l+2) = \mathbf{s}'(l+2)$
From [2] and [5], we have
(11) $\mathsf{x}''[l+2..j] = \mathsf{x}'[l+2..j]$
By Lemma 27 on [10] and [11], we have
(12) $\mathbf{s}''(j+1) = \mathbf{s}'(j+1)$
From [4] and [12], we have
$\mathbf{s}''(j+1) = \mathbf{s}(j+1)$

Lemma 32. *In every execution, if a request r at a position j $\mathcal{S}$-commutes with preceding requests R from position i, then shifting right R by one position and moving r to i keeps the post-state of j the same.*

Formally, for every context $\mathbf{c}$ and execution x, for every position i and j such that $0 \leq i < j < |\mathcal{R}(\mathsf{x})|$, if

- *for every position k, $i \leq k < j$, $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k)) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}(j))$,*
- *let x' be $\mathsf{x}[i \mapsto \mathsf{x}(j)][i+1 \mapsto \mathsf{x}(i)]..[j \mapsto \mathsf{x}(j-1)]$, and*
- *let $\mathbf{s}$ and $\mathbf{s}'$ be the state functions for x and x',*

then $\mathbf{s}'(j+1) = \mathbf{s}(j+1)$.

Proof.
Similar to Lemma 31.

Lemma 33. *In every $\mathcal{S}$-conflict-synchronizing replicated execution xs, for every request r, the requests that precede r in the execution $\mathsf{xs}(n_1)$ of a node n_1 but do not precede r in the execution $\mathsf{xs}(n_2)$ of another node n_2 can be moved right in $\mathsf{xs}(n_1)$ to a block before r and the pre-state of r remains the same.*

Formally, for every context $\mathbf{c}$ and $\mathcal{S}$-conflict-synchronizing replicated execution xs, for every request r in R, and pair of nodes n_1 and n_2,

- *let $P(n)$ be $\{r' \mid r' \prec_{\mathsf{xs}(n)} r\}$*
- *let i be $\mathsf{xs}(n_1)^{-1}(r)$*
- *let x' be $\mathsf{xs}(n_1)|_{P(n_1)\cap P(n_2)} \cdot \mathsf{xs}(n_1)|_{P(n_1)\setminus P(n_2)}\ [i \mapsto r]$*
- *let $\mathbf{s}$ and $\mathbf{s}'$ be the state functions for $\mathsf{xs}(n_1)$ and x',*

$\mathbf{s}'(i) = \mathbf{s}(i)$.

Proof.
We prove the following stronger statement.

For every context $\mathbf{c}$ and $\mathcal{S}$-conflict-synchronizing replicated execution xs, for every request r in R, and pair of nodes n_1 and n_2,

- let $P(n)$ be $\{r' \mid r' \prec_{\mathsf{xs}(n)} r\}$
- let i be $\mathsf{xs}(n_1)^{-1}(r)$

for every $0 \leq k \leq |P(n_1) \setminus P(n_2)|$,

- let R be the k rightmost requests of $\mathsf{xs}(n_1)$ in $P(n_1) \setminus P(n_2)$,
- let x' be $\mathsf{xs}(n_1)|_{P(n_1)\setminus R} \cdot \mathsf{xs}(n_1)|_R\ [i \mapsto r]$
- let $\mathbf{s}$ and $\mathbf{s}'$ be the state functions for $\mathsf{xs}(n_1)$ and x',

$\mathbf{s}'(i) = \mathbf{s}(i)$.

The original statement is derived by setting k to $|P(n_1) \setminus P(n_2)|$.
$R = P(n_1) \setminus P(n_2)$ and $P(n_1) \setminus R = P(n_1) \cap P(n_2)$.

We assume that
(1) xs is a $\mathcal{S}$-conflict-synchronizing replicated execution.
(2) $P(n) = \{r' \mid r' \prec_{\mathsf{xs}(n)} r\}$
(3) $i = \mathsf{xs}(n_1)^{-1}(r)$
(4) $0 \leq k \leq |P(n_1) \setminus P(n_2)|$,
(5) R is the k rightmost requests of $\mathsf{xs}(n_1)$ in $P(n_1) \setminus P(n_2)$,
(6) $\mathsf{x}' = \mathsf{xs}(n_1)|_{P(n_1)\setminus R} \cdot \mathsf{xs}(n_1)|_R\ [i \mapsto r]$
(7) $\mathbf{s}$ and $\mathbf{s}'$ are the state functions for $\mathsf{xs}(n_1)$ and x',
We prove that
$\mathbf{s}'(i) = \mathbf{s}(i)$.

Proof by induction on k.
Base case:
$k = 0$
Trivial.
Inductive case:
We assume that
(8) $\mathbf{s}'(i) = \mathbf{s}(i)$.
(9) $R' = R \cup \{r'\}$ is the $k+1$ rightmost requests of $\mathsf{xs}(n_1)$ in $P(n_1) \setminus P(n_2)$,
(10) $\mathsf{x}'' = \mathsf{xs}(n_1)|_{P(n_1)\setminus R'} \cdot \mathsf{xs}(n_1)|_{R'}\ [i \mapsto r]$
(11) $\mathbf{s}''$ is the state function for x'',
We prove that
$\mathbf{s}''(i) = \mathbf{s}(i)$.

Let
(12) $j = \mathsf{x}'^{-1}(r')$

By Def. 2

(13) For every k that $j < k$,
$\mathsf{xs}(n_1)(j) \prec_{\mathsf{xs}(n_1)} \mathsf{xs}(n_1)(k)$

From [6], [5], [9] and [12]
(14) For every k that $j < k \leq |P(n_1) \setminus R| - 1$,
$\mathsf{xs}(n_1)(k) \in P(n_1) \wedge$
$\mathsf{xs}(n_1)(k) \notin P(n_1) \setminus P(n_2)$
(15) $j = \mathsf{xs}(n_1)^{-1}(r')$
From [14]
(16) For every k that $j < k \leq |P(n_1) \setminus R| - 1$,
$\mathsf{xs}(n_1)(k) \in P(n_1) \cap P(n_2)$
From [16] and [2]
(17) For every k that $j < k \leq |P(n_1) \setminus R| - 1$,
$\mathsf{xs}(n_1)(k) \prec_{\mathsf{xs}(n_2)} r$

From [9], [15] and [2]
(18) $\mathsf{xs}(n_1)(j) \not\prec_{\mathsf{xs}(n_2)} r$

By Lemma 28 on [1], [13], [17] and [18]
(19) For every k that $j < k \leq |P(n_1) \setminus R| - 1$,
$\mathsf{call_c}(\mathsf{xs}(n_1)(j)) \leftrightarrows_{\mathcal{S}} \mathsf{call_c}(\mathsf{xs}(n_1)(k))$

By Lemma 31 on [19] and [6]
(20) let x''' be $\mathsf{x}'[j \mapsto \mathsf{x}'[j+1]]$ ..
$[|P(n_1) \setminus R| - 2 \mapsto \mathsf{x}'[|P(n_1) \setminus R| - 1]]$
$[|P(n_1) \setminus R| - 1 \mapsto \mathsf{x}'[j]] \cdot$
$\mathsf{x}'|_R\ [i \mapsto r]$
(21) let $\mathbf{s}'''$ be the state function for x'''
(22) $\mathbf{s}'''(|P(n_1) \setminus R|) = \mathbf{s}'(|P(n_1) \setminus R|)$
From [20] and [15],
(23) $\mathsf{x}''' = \mathsf{x}'[j \mapsto \mathsf{x}'[j+1]]$ ..
$[|P(n_1) \setminus R| - 2 \mapsto \mathsf{x}'[|P(n_1) \setminus R| - 1]]$
$[|P(n_1) \setminus R| - 1 \mapsto r'] \cdot$
$\mathsf{x}'|_R\ [i \mapsto r]$
From [6] and [23],
(24) $\mathsf{x}''' = \mathsf{xs}(n_1)|_{P(n_1) \setminus R \cup \{r'\}} \cdot$
$\mathsf{xs}(n_1)|_{\{r'\} \cup R}$
$[i \mapsto r]$
From [24], [9], [10],
(25) $\mathsf{x}''' = \mathsf{x}''$
From [25], [11], [21],
(26) $\mathbf{s}''' = \mathbf{s}''$
From [22] and [26],
(27) $\mathbf{s}''(|P(n_1) \setminus R|) = \mathbf{s}'(|P(n_1) \setminus R|)$
From [6], [10] and [9],

(28) $\mathsf{x}''[|P(n_1) \setminus R| \,..\, i-1] = x'[|P(n_1) \setminus R| \,..\, i-1]$
By Lemma 27 on [27] and [28]
(29) $\mathbf{s}''(i) = \mathbf{s}'(i)$
From [8] and [29]
$\mathbf{s}''(i) = \mathbf{s}(i)$

Lemma 34. *In every execution x, if the request r at position j $\mathcal{P}$-L-commutes with the immediately preceding requests R and r is permissible in x then removing R from x keeps r permissible.*

Formally, for every context $\mathbf{c}$ and execution x, for every position i and j such that $0 \leq i < j < |\mathcal{R}(\mathsf{x})|$, if

- *for every position k, $i \leq k < j$, $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(j)) \leftarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}(k))$,*
- *$\mathcal{P}(\mathbf{c}, \mathsf{x}, \mathsf{x}(j))$*
- *let x' be $(\mathsf{x} \setminus [i..j])[i \mapsto \mathsf{x}(j)]$,*

then $\mathcal{P}(\mathbf{c}, \mathsf{x}', \mathsf{x}'(i))$

Proof.
We prove the following stronger statement.

For every context $\mathbf{c}$ and execution x, for every position i and j such that $0 \leq i < j < |\mathcal{R}(\mathsf{x})|$, if

- for every position k, $i \leq k < j$, $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(j)) \leftarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}(k))$,
- $\mathcal{P}(\mathbf{c}, \mathsf{x}, \mathsf{x}(j))$

for every $i \leq k \leq j$,

- let x' be $(\mathsf{x} \setminus [k..j])[k \mapsto \mathsf{x}(j)]$,

then $\mathcal{P}(\mathbf{c}, \mathsf{x}', \mathsf{x}'(k))$

The original statement is derived by setting k to i.

We assume that
(1) for every k, $i \leq k < j$, $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(j)) \leftarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}(k))$,
(2) $\mathcal{P}(\mathbf{c}, \mathsf{x}, \mathsf{x}(j))$
(3) $\mathsf{x}' = (\mathsf{x} \setminus [k..j])[k \mapsto \mathsf{x}(j)]$,
We prove that
$\mathcal{P}(\mathbf{c}, \mathsf{x}', \mathsf{x}'(k))$

Proof by induction on k from j down to i.
(equivalently $k' = j - k$ from 0 to $j - i$)
Base case:
$k = j$
Trivial.
Inductive case:

We assume that
(4) $\mathcal{P}(\mathbf{c}, \mathsf{x}', \mathsf{x}'(k))$
(5) $\mathsf{x}'' = (\mathsf{x} \setminus [k-1..j])[k-1 \mapsto \mathsf{x}(j)]$,
We prove that
$\mathcal{P}(\mathbf{c}, \mathsf{x}'', \mathsf{x}''(k-1))$

Let
(6) $\mathbf{s}$, $\mathbf{s}'$ and $\mathbf{s}''$ be the state functions for x, x' and x''.

From [3]
(7) $\mathsf{x}'[0..k-1] = \mathsf{x}[0..k-1]$
By Lemma 27 on [7]
(8) $\mathbf{s}'(k-1) = \mathbf{s}(k-1)$

From Def. 11 and Def. 3 on [4]
(9) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(\mathsf{x}'(k)))(\mathbf{s}'(k))$
(10) $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(\mathsf{x}(j)))(\mathbf{s}'(k)))$
From [9] and [3]
(11) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(\mathsf{x}(j)))(\mathbf{s}'(k))$

By Def. 8 on [3] and [6],
(12) $\mathbf{s}'(k) = \mathsf{update}(\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k-1)))(\mathbf{s}'(k-1))$

By [1] on $k-1$
(13) $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(j)) \leftarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}(k-1))$

By Def. 8 and and Def. 3 on [13], [12], [11],
and [10],
(14) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(\mathsf{x}(j)))(\mathbf{s}'(k-1))$
(15) $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(\mathsf{x}(j)))(\mathbf{s}'(k-1)))$

From [5]
(16) $\mathsf{x}''[0..k-2] = \mathsf{x}[0..k-2]$
By Lemma 27 on [16]
(17) $\mathbf{s}''(k-1) = \mathbf{s}(k-1)$
From [8] and [17]
(18) $\mathbf{s}''(k-1) = \mathbf{s}'(k-1)$

From [18], [14] and [15]
(19) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(\mathsf{x}(j)))(\mathbf{s}''(k-1))$
(20) $\mathcal{I}(\mathsf{call}_{\mathbf{c}}(\mathsf{update}(\mathsf{x}(j)))(\mathbf{s}''(k-1)))$

By Def. 8 on [5] and [6],
(21) $\mathsf{x}''(k-1) = \mathsf{x}(j)$
From [19], [20], and [21]

(22) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(\mathsf{x}''(k-1)))(\mathbf{s}''(k-1))$

From Def. 3 and Def. 11 on [22] and [20]
$\mathcal{P}(\mathbf{c}, \mathsf{x}'', \mathsf{x}''(k-1))$

Lemma 35. *In every locally-permissible, $\mathcal{S}$-conflict-synchronizing and dependency-preserving replicated execution xs, for every request r such that $\mathsf{call}_{\mathbf{c}}(r)$ is not invariant-sufficient, let n_1 be its originating node $\mathsf{orig}_{\mathbf{c}}(r)$, let R be the requests that precede r in the execution $\mathsf{xs}(n_1)$ but do not precede r in the execution $\mathsf{xs}(n_2)$ of another node n_2, R can be removed from $\mathsf{xs}(n_1)$ and r remains permissible.*

Formally, for every context $\mathbf{c}$ and locally-permissible, $\mathcal{S}$-conflict-synchronizing and dependency-preserving replicated execution xs, for every request r in R such that $\mathsf{call}_{\mathbf{c}}(r)$ is not invariant-sufficient, and node n_2,

- *let n_1 be $\mathsf{orig}_{\mathbf{c}}(r)$*
- *let $P(n)$ be $\{r' \mid r' \prec_{\mathsf{xs}(n)} r\}$*
- *let x' be $\mathsf{xs}(n_1)|_{P(n_1)\cap P(n_2)}\ [|P(n_1) \cap P(n_2)| \mapsto r]$*

then $\mathcal{P}(\mathbf{c}, \mathsf{x}', r)$.

Proof.
We assume that
(1) xs is a locally-permissible replicated execution
(2) xs is a $\mathcal{S}$-conflict-synchronizing replicated execution
(3) xs is a dependency-preserving replicated execution
(4) $\mathsf{call}_{\mathbf{c}}(r)$ is invariant-sufficient.
(5) $n_1 = \mathsf{orig}_{\mathbf{c}}(r)$
(6) $P(n) = \{r' \mid r' \prec_{\mathsf{xs}(n)} r\}$
(7) $\mathsf{x}' = \mathsf{xs}(n_1)|_{P(n_1)\cap P(n_2)}\ [|P(n_1) \cap P(n_2)| \mapsto r]$
We prove that
$\mathcal{P}(\mathbf{c}, \mathsf{x}', r)$

By Def. 12 and Def. 11 on [1] and [5]
(8) $\mathcal{P}(\mathbf{c}, \mathsf{xs}(n_1), r)$,

By Lemma 33 on [2] and [6],
(9) let i be $\mathsf{xs}(n_1)^{-1}(r)$
(10) let x'' be $\mathsf{xs}(n_1)|_{P(n_1)\cap P(n_2)} \cdot \mathsf{xs}(n_1)|_{P(n_1)\setminus P(n_2)} [i \mapsto r]$
(11) let $\mathbf{s}$ and $\mathbf{s}''$ be the state functions for $\mathsf{xs}(n_1)$ and x'',
(12) $\mathbf{s}''(i) = \mathbf{s}(i)$.

By Lemma 30 on [3], [5], [6] and [4]
(13) for every r' in $P(n_1) \setminus P(n_2)$, $\mathsf{call}_{\mathbf{c}}(r) \leftarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r')$
From [10] and [13]

(14) for every position k, $|P(n_1) \cap P(n_2)| \leq k < i$,
$\mathsf{call}_{\mathbf{c}}(\mathsf{x}''(i)) \leftarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}''(k))$,

From [8] and [9]
(15) $\mathcal{P}(\mathbf{c}, \mathsf{xs}(n_1), \mathsf{xs}(n_1)(i))$,
By Def. 11 and Def. 8 on [11], [12] and [15]
(16) $\mathcal{P}(\mathbf{c}, \mathsf{x}'', \mathsf{x}''(i))$,

By Lemma 34 on [14] and [16]
(17) let x''' be $(\mathsf{x}'' \setminus [|P(n_1) \cap P(n_2)|..i])$
$[|P(n_1) \cap P(n_2)| \mapsto \mathsf{x}''(i)]$,
(18) $\mathcal{P}(\mathbf{c}, \mathsf{x}''', \mathsf{x}'''(|P(n_1) \cap P(n_2)|))$

From [17] and [10]
(19) $\mathsf{x}''' = \mathsf{xs}(n_1)|_{P(n_1) \cap P(n_2)}\ [|P(n_1) \cap P(n_2)| \mapsto r]$
From [7] and [19]
(20) $\mathsf{x}''' = \mathsf{x}'$
From [18], [20], [7]
$\mathcal{P}(\mathbf{c}, \mathsf{x}', r)$

Lemma 36. *In every execution x, if a request r is $\mathcal{P}$-R-commutative with a sequence of requests R and is permissible in the state immediately before them, it is permissible in the state immediately after them as well.*

Formally, for every context $\mathbf{c}$, execution x and request r, for every position i and j such that $0 \leq i < j < |\mathcal{R}(\mathsf{x})|$, if

- *for every position k, $i \leq k \leq j$, $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k)) \rightarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r)$,*
- *let $\mathbf{s}$ be the state function for x*
- *$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(i))$,*
- *$\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(i)))$,*

then

- *$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(j+1))$,*
- *$\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(j+1)))$.*

Proof.
We prove the following stronger statement.

For every context $\mathbf{c}$, execution x and request r, for every position i and j such that $0 \leq i < j < |\mathcal{R}(\mathsf{x})|$, if

- for every position k, $i \leq k \leq j$, $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k)) \rightarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r)$,
- let $\mathbf{s}$ be the state function for x
- $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(i))$,
- $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(i)))$,

for every k, $i \leq k \leq j+1$,

- $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k))$,
- $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k)))$.

The original statement is derived by setting k to $j + 1$.

We assume that
(1) for every position k, $i \leq k \leq j$,
$\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k)) \rightarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r)$,
(2) let $\mathbf{s}$ be the state function for x
(3) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(i))$,
(4) $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(i)))$,
We prove that
$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k))$,
$\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k)))$.

Proof by induction on k from i to $j + 1$.
Base case:
$k = i$
Trivial.
Inductive case:
We assume that
(5) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k))$
(6) $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k)))$
We prove that
$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k+1))$
$\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k+1)))$

By Def. 8, we have
(7) $\mathbf{s}(k+1) = \mathsf{update}(\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k)))(\mathbf{s}(k))$
From [1]
(8) $\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k)) \rightarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r)$
By Def. 5 and Def. 3 on [8], [5], [6], [7]
$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k+1))$
$\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}(k+1)))$

Lemma 37. *For every pair of executions x and x' of a set of requests R in a context $\mathbf{c}$, if*

- *for every r and r' in R, if $\mathsf{call}_{\mathbf{c}}(r) \not\rightleftarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(r')$ and $r \prec_{\mathsf{x}} r'$, then $r \prec_{\mathsf{x}'} r'$*
- *let $\mathbf{s}$ and $\mathbf{s}'$ be the state functions of x and x'*

then $\mathbf{s}(|R|) = \mathbf{s}'(|R|)$.

Proof.
We prove the following stronger statement.

For every pair of executions x and x' of a set of requests R in a context $\mathbf{c}$, if

- for every r and r' in R, if $\mathsf{call}_{\mathbf{c}}(r) \not\rightleftarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(r')$ and $r \prec_{\mathsf{x}} r'$, then $r \prec_{\mathsf{x}'} r'$
- let $\mathbf{s}$ and $\mathbf{s}'$ be the state functions of x and x'

for every k, $0 \leq k < |R| + 1$, there exists an execution x_k on R such that

- if $k = 0$
  $\mathsf{x}_k = \mathsf{x}'$
  else
  $\mathsf{x}_k^1 = \mathsf{x}[0..k-1]$,
  $\mathsf{x}_k^2 = \mathsf{x}'|_{R \setminus \mathcal{R}(\mathsf{x}[0..k-1])}$,
  $\mathsf{x}_k = \mathsf{x}_k^1 \cdot \mathsf{x}_k^2$,
  let $\mathbf{s}_k$ be the state function of x_k,
- $\mathbf{s}_k(|R|) = \mathbf{s}'(|R|)$

The original statement is derived by setting k to $|R|$.

Proof by induction on k.
Base case: $k = 0$
Trivial.

Inductive case:
We assume that
(1) for every r and r' in R,
if $\mathsf{call}_{\mathbf{c}}(r) \not\rightleftarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(r')$ and $r \prec_{\mathsf{x}} r'$, then $r \prec_{\mathsf{x}'} r'$
(2) let $\mathbf{s}$ and $\mathbf{s}'$ be the state functions of x and x'
(3) $\mathsf{x}_k^1 = \mathsf{x}[0..k-1]$,
(4) $\mathsf{x}_k^2 = \mathsf{x}'|_{R \setminus \mathcal{R}(\mathsf{x}[0..k-1])}$,
(5) $\mathsf{x}_k = \mathsf{x}_k^1 \cdot \mathsf{x}_k^2$,
(6) let $\mathbf{s}_k$ be the state function of x_k,
(7) $\mathbf{s}_k(|R|) = \mathbf{s}'(|R|)$
(8) $\mathsf{x}_{k+1}^1 = \mathsf{x}[0..k]$,
(9) $\mathsf{x}_{k+1}^2 = \mathsf{x}'|_{R \setminus \mathcal{R}(\mathsf{x}[0..k])}$,
(10) $\mathsf{x}_{k+1} = \mathsf{x}_{k+1}^1 \cdot \mathsf{x}_{k+1}^2$,
(11) let $\mathbf{s}_{k+1}$ be the state function of x_{k+1},
We prove that
$\mathbf{s}_{k+1}(|R|) = \mathbf{s}'(|R|)$

(12) let r be $\mathsf{x}(k)$
(13) let i be $\mathsf{x}_k^{-1}(r)$
From [3], [4], and [5],
(14) for every l, $k \leq l < i$
$\mathsf{x}_k(l) \prec_{\mathsf{x}'} \mathsf{x}_k(i)$
(15) for every l, $k \leq l < i$
$\mathsf{x}_k(l) \notin \mathcal{R}(\mathsf{x}[0..k-1])$

From [15]
(16) for every l, $k \leq l < i$
$\mathsf{x}_k(l) \not\prec_{\mathsf{x}} \mathsf{x}(k)$
From [16], [12] and [13]
(17) for every l, $k \leq l < i$
$\mathsf{x}_k(l) \not\prec_{\mathsf{x}} \mathsf{x}_k(i)$
From [1], [14] and [17]
(18) for every l, $k \leq l < i$
$\mathsf{call}_{\mathbf{c}}(\mathsf{x}_k(l)) \not\Rightarrow_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(\mathsf{x}_k(i))$

By Lemma 32 on [3], [4], [5], [18]
(19) $\mathsf{x}_k^{1\prime} = \mathsf{x}[0..k][k \mapsto \mathsf{x}_k(i)]$,
(20) $\mathsf{x}_k^{2\prime} = \mathsf{x}'|_{R \setminus [\mathcal{R}(\mathsf{x}[0..k-1]) \cup \mathsf{x}_k(i)]}$,
(21) $\mathsf{x}'_k = \mathsf{x}_k^{1\prime} \cdot \mathsf{x}_k^{2\prime}$,
(22) let $\mathbf{s}'_k$ be the state function of x'_k,
(23) $\mathbf{s}'_k(i+1) = \mathbf{s}_k(i+1)$

From [19], [20], [21], [3], [4], [5],
(24) $\mathsf{x}'_k[i+1..|R|-1] = \mathsf{x}_k[i+1..|R|-1]$
By Lemma 27 on [23], [24], [6], [22],
(25) $\mathbf{s}'_k(|R|) = \mathbf{s}_k(|R|)$

From [19], [12], [13] and [8]
(26) $\mathsf{x}_k^{1\prime} = \mathsf{x}_{k+1}^1$,
From [20], [12], [13] and [9]
(27) $\mathsf{x}_k^{1\prime} = \mathsf{x}_{k+1}^1$,
From [26], [27], [21] and [10]
(28) $\mathsf{x}'_k = \mathsf{x}_{k+1}$,
From [28], [22], and [11]
(29) $\mathbf{s}'_k = \mathbf{s}_{k+1}$,
From [25], and [29]
(30) $\mathbf{s}_{k+1}(|R|) = \mathbf{s}_k(|R|)$

From [7], and [30]
$\mathbf{s}_{k+1}(|R|) = \mathbf{s}'(|R|)$

Lemma 38. *Every $\mathcal{S}$-conflict-synchronizing complete replicated execution is convergent.*

Proof.
Immediate from Def. 14, Lemma 37, and Def. 9.

Lemma 39. *Every well-coordinated replicated execution is permissible. Every replicated execution that is locally-permissible, conflict-synchronizing, and dependency-preserving is permissible.*

Proof.
We assume that
(1) xs is a replicated execution of a context $\mathbf{c}$.
(2) xs is locally-permissible.
(3) xs is conflict-synchronizing.
(4) xs is dependency-preserving.
We prove that
xs is permissible.
By Def. 11, we need to show that
for every n_1 in $\mathcal{N}$ and r in R,
$\mathcal{P}(\mathbf{c}, \mathsf{xs}(n_1), r)$
that is
(5) let $\mathbf{s}_1$ be the state function of $\mathsf{xs}(n_1)$,
(6) let i be $\mathsf{xs}(n_1)^{-1}(r)$
then
$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}_1(i))$
$\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}_1(i)))$

Induction on a linear extension of $\mathsf{hb}_{\mathsf{xs}}$:
The induction hypothesis is that
(7) For every n' and r',
if (n', r') $\mathsf{hb}_{\mathsf{xs}}$ (n, r),
then $\mathcal{P}(\mathbf{c}, \mathsf{xs}(n'), r')$

By Lemma 25, on [3]
(8) xs is $\mathcal{S}$-conflict-synchronizing.
(9) xs is $\mathcal{P}$-conflict-synchronizing.

We consider two cases:
Case:
(10) $\mathsf{call}_c(r)$ is invariant-sufficient.
We first show that
(11) $\mathcal{I}(\mathbf{s}_1(i))$
Let
(12) $r' = \mathsf{xs}(n_1)(i-1)$
From Def. 3, we have
(13) $(n_1, r') \prec_{hb_{\mathsf{xs}}} (n_1, r)$
From [7] and [13],
(14) $\mathcal{P}(\mathbf{c}, \mathsf{xs}(n_1), r')$
By Def. 11 on [14] and [12]
(15) $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(\mathsf{xs}(n_1)(i-1)))(\mathbf{s}_1(i)))$
By Def. 8 on [15] and [5]
$\mathcal{I}(\mathbf{s}_1(i))$

By Def. 4 and Def. 3 on [10], [6] and [11]

$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}_1(i))$
$\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}_1(i)))$

Case:
(16) $\mathsf{call}_c(r)$ is not invariant-sufficient.
Now, we consider two nested cases:
Case: $n_1 = \mathsf{orig}_{\mathbf{c}}(r)$
Immediate from [2] and Def. 12.

Case: $n_1 \neq \mathsf{orig}_{\mathbf{c}}(r)$
By Lemma 35 on [2], [8], [4], and [16]
(17) let n_2 be $\mathsf{orig}_{\mathbf{c}}(r)$
(18) let $P(n)$ be $\{r' \mid r' \prec_{\mathsf{xs}(n)} r\}$
(19) let x'_2 be $\mathsf{xs}(n_2)|_{P(n_2)\cap P(n_1)}$
$[|P(n_2) \cap P(n_1)| \mapsto r]$
(20) $\mathcal{P}(\mathbf{c}, \mathsf{x}'_2, r)$.

By Lemma 33 on [8], [18], and [6]
(21) let x'_1 be $\mathsf{xs}(n_1)|_{P(n_1)\cap P(n_2)} \cdot$
$\mathsf{xs}(n_1)|_{P(n_1)\setminus P(n_2)}$
$[i \mapsto r]$.
(22) let $\mathbf{s}'_1$ be the state function x'_1.
(23) $\mathbf{s}'_1(i) = \mathbf{s}_1(i)$.

By Def. 14 on [8]
(24) for every r and r' in $P(n_1) \cap P(n_2)$,
if $\mathsf{call}_{\mathbf{c}}(r) \leftrightarrows_{\mathcal{S}} \mathsf{call}_{\mathbf{c}}(r')$ and $r \prec_{\mathsf{xs}(n_1)} r'$
then $r \prec_{\mathsf{xs}(n_2)} r'$
By Lemma 37 on [24], [21], [22], and [19]
(25) let $\mathbf{s}'_2$ be the state function x'_2.
(26) $\mathbf{s}'_1(|P(n_1) \cap P(n_2)|) = \mathbf{s}'_2(|P(n_1) \cap P(n_2)|)$
By Def. 11 on [20], [19] and [25]
(27) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}'_2(|P(n_1) \cap P(n_2)|))$
(28) $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}'_2(|P(n_1) \cap P(n_2)|)))$
From [26], [27], and [28]
(29) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}'_1(|P(n_1) \cap P(n_2)|))$
(30) $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}'_1(|P(n_1) \cap P(n_2)|)))$

By Lemma 29 on [9], [18] and [16]
(31) for every r' in $P(n_1) \setminus P(n_2)$,
$\mathsf{call}_{\mathbf{c}}(r') \rightarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r)$
From [31] and [21]
(32) for every position k, $|P(n_1) \cap P(n_2)| \leq k \leq i-1$,
$\mathsf{call}_{\mathbf{c}}(\mathsf{x}(k)) \rightarrow_{\mathcal{P}} \mathsf{call}_{\mathbf{c}}(r)$,

By Lemma 36 on [21], [32], [22], [29] and [30]
(33) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}'_1(i))$
(34) $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}'_1(i)))$
From [33], [34] and [23]
(35) $\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}_1(i))$
(36) $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}_1(i)))$
By Def. 8 on [36], [5] and [6]
(37) $\mathcal{I}(\mathbf{s}_1(i+1))$
The conclusion is [35] and [37].

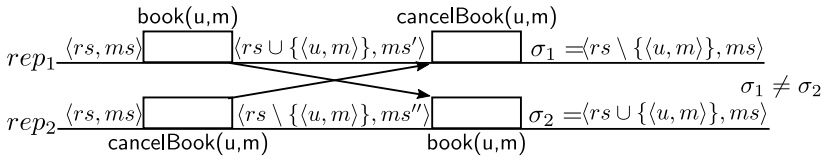
Lemma 40. *Every replicated execution that is locally-permissible, conflict-synchronizing, and dependency-preserving has integrity.*

Proof.
We assume that
(1) xs is a replicated execution of a context $\mathbf{c}$.
(2) xs is locally-permissible,
xs is conflict-synchronizing,
xs is dependency-preserving
We prove that
xs has integrity.
By Def. 11, we need to show that
for every n_1 in $\mathcal{N}$ and r in R,
$\mathsf{integrity}(\mathbf{c}, \mathsf{xs}(n_1), r)$
that is
(3) let $\mathbf{s}_1$ be the state function of $\mathsf{xs}(n_1)$,
(4) let i be $\mathsf{xs}(n_1)^{-1}(r)$
then
$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}_1(i))$
$\mathcal{I}(\mathbf{s}_1(i))$

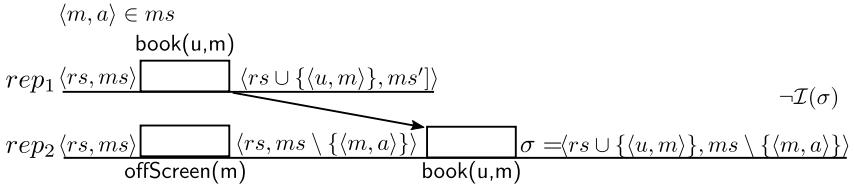
From Lemma 39 on [1] and [2], we have
$\mathsf{guard}(\mathsf{call}_{\mathbf{c}}(r))(\mathbf{s}_1(i))$

If $i = 0$, then from Def. 1, we have
$\mathcal{I}(\mathbf{s}_1(i))$
Otherwise,
From Lemma 39 on [1] and [2], we have
(5) $\mathcal{I}(\mathsf{update}(\mathsf{call}_{\mathbf{c}}(\mathsf{xs}(n)(i-1)))(\mathbf{s}_1(i-1)))$
From Def. 8 on [3] and [5]
$\mathcal{I}(\mathbf{s}_1(i))$

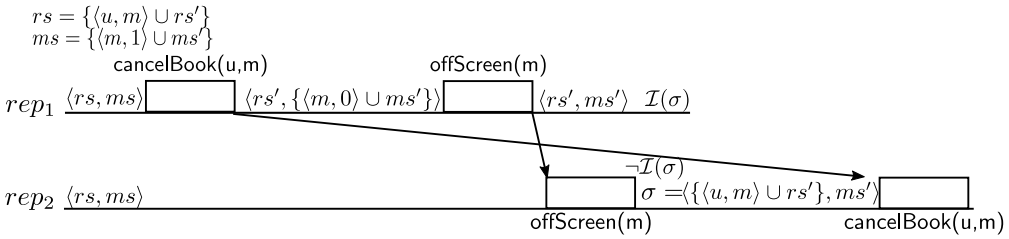
4 Coordination Conditions



(a) $\mathcal{S}$-conflict

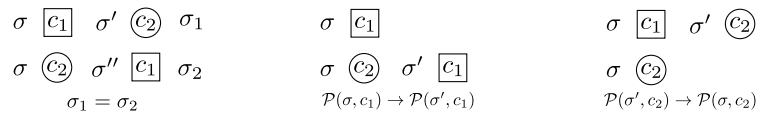


(c) $\mathcal{P}$-conflict



(e) Dependence

Fig. 3: Incorrect Executions for the Movie Booking Use-case.



(a) $c_1 \leftrightarrows_{\mathcal{S}} c_2$ State-Commutativity

(b) $c_1 \to_{\mathcal{P}} c_2$ $\mathcal{P}$-R-Commutativity

(c) $c_2 \leftarrow_{\mathcal{P}} c_1$ $\mathcal{P}$-L-Commutativity

Fig. 4: Coordination Avoidance Conditions. Square and circle around method calls are just visual aids to see the movements.

(a) (b)

Fig. 5: (a) $\mathcal{S}$-conflict graph and (b) $\mathcal{P}$-conflict graph for the movie use-case

	book	cancelOrder	offScreen	specialReserve	increaseSpace
book	×	×	✓	✓	✓
cancelOrder	×	✓	✓	✓	✓
offScreen	✓	✓	✓	✓	✓
specialReserve	✓	✓	✓	✓	✓
increaseSpace	✓	✓	✓	✓	✓

(b) $\mathcal{S}$-commute

	book	cancelOrder	offScreen	specialReserve	increaseSpace
book	×	✓	×	×	✓
cancelOrder	✓	✓	✓	✓	✓
offScreen	×	✓	✓	✓	✓
specialReserve	×	✓	✓	×	✓
increaseSpace	✓	✓	✓	✓	✓

(c) $\mathcal{P}$-concur

	book	cancelOrder	offScreen	specialReserve	increaseSpace
book	✓	×	✓	✓	×
cancelOrder	✓	✓	✓	✓	✓
offScreen	✓	×	✓	✓	✓
specialReserve	✓	×	✓	✓	×
increaseSpace	✓	✓	✓	✓	✓

(e) Independent

Fig. 6: The coordination conditions for the movie booking Use-case

	deposit	withdraw	query
deposit	✓	✓	✓
withdraw	✓	✓	✓
query	✓	✓	✓

(b) $\mathcal{S}$-commute

	deposit	withdraw	query
deposit	✓	✓	✓
withdraw	✓	×	✓
query	✓	✓	✓

(c) $\mathcal{P}$-concur

	deposit	withdraw	query
deposit	✓	✓	✓
withdraw	×	✓	✓
query	✓	✓	✓

(e) Independent

Fig. 7: The coordination conditions for the bank account use-case

5 Protocol

In this section, we present the protocol that implements a recency-aware replicated object. Given an object definition, the protocol benefits from both static and dynamic coordination analysis to guarantee convergence, integrity and recency.

This protocol builds on top of the Hamsaz basic blocking replication protocol that guarantees integrity and convergence but not recency. In the basic protocol, to maintain a total order between conflicting calls, one method in every pair of conflicting methods performs the synchronization. Consider a pair of conflicting methods m and m'. We let calls on m avoid synchronization and calls on m' perform synchronization. When the user requests to call m', the originating replica reaches out to other replicas and blocks the execution of calls on m. Then, replicas update others with the calls on m that they have executed. Once the replicas apply these updates, they have executed the same set of calls on m. Then, the call on m' is executed at all replicas. Hence, the total order between calls to m and m' is preserved. Finally, calls on m are unblocked. In a general conflict graph, each method in the minimum vertex cover of the graph needs to perform the synchronization

The recency-aware coordination protocol is presented in Fig. 8. It accepts requests $\mathsf{call}(c)$ to execute the call c and in return issues an indication $\mathsf{ret}(c, v)$ with the return value v or issues an indication that c is aborted $\mathsf{aborted}(c)$. The parameters to the protocol are the following information that result from static coordination analysis: the staleness bound ϵ, and the set SConf of methods that have state-conflicts. The protocol uses four sub-protocols: the reliable broadcast *rb*, the perfect point-to-point link *pl*, the perfect failure detector *pfd*, and the basic replication object *bro*. The protocol stores the following state at each replica: the user-defined state of the object σ, the composition of the calls executed locally *buff*, a queue *wq* which stores the calls that are issued but cannot be executed yet to maintain recency, the set of correct replicas *up*, and the mapping p from each replica to the set of pending calls for that replica.

Upon a request to execute a call c (at R_0), it is first checked for permissibly (at R_1). The call is aborted if it is not permissible. (The guard of the request handler at R_0 delays methods that are blocked by the underlying sub-protocol *bro*.) Otherwise, the conditions of the rule CallLocal of the operational semantics (§ 4) are checked to decide whether the call can be executed locally and buffered. To check for all-state-commutativity of the call c (at R_4), we check if the method is not in the SConf. Also, we dynamically check the invariant-sufficiency and let-$\mathcal{P}$-R-commutativity of the buffer after the call c is added (at R_4). The decision of the validity of the conditions is dispatched to the solver (at D_1). If all the conditions for local execution hold, the temporary pending map p' is calculated by updating the pending map p with the new call c added to the buffer (at R_5-R_6). In order to execute the call locally, the bound is also checked. If the $\mathsf{InBound}$ function returns true, the pending map p is updated with p' (at R_8), the call is executed locally and the buffer is updated by composing the current call with it (at R_9). To execute a call (at E_0), the updated state is stored, the return value v is calculated (at E_1), and a return indication with the value v is issued

```
RecencyAwareObject
  request: call(C)
  indication: ret(C, V) | aborted(C)
  Params:
    ϵ: Int
    SConf: Set[M]
  Using:
    rb: ReliableBroadcast
    pl: PerfectPointToPointLink
    pfd: Perfect Failure Detector
    bro: BasicRepObject
  State:
    σ: Σ = σ0; buff = ∅; wq = ∅;
    up = N;  p: N ↦ Set[C] = N → ∅

R0  request (call(c))   if (method(c) ∉ blocked(bro))
R1    if (¬P(σ, c))
R2      issue indication aborted(c)
R3    else
R4      if (method(c) ∉ SConf ∧
              dynamicCheck(c · buff))
R5        foreach (r ∈ up \ {self})
R6          p'(r) ← ((p(r) \ {buff}) ∪ {c · buff})
R7        if (InBound(p'))
R8          p ← p'
R9          exec(c);  buff  ← c · buff
R20       else
R11         issue request (rb, broadcast(buff(buff)))
R12         insert(wq, c)
R13     else
R14       foreach (r ∈ up \ {self})
R15         p'(r) ← (p(r) ∪ {c})
R16       if (InBound(p'))
R17         p ← p'
R18         issue request(bro, call(c))
R19       else
R20         issue request (rb, broadcast(buff(buff)))
R21         insert(wq, c)

F0  indication crash(pfd, p)
F1    up ← up \ {p}

I0  fun InBound(p)
I1    foreach(n ∈ up)
I2      if (∑_{c'∈p(n)} weight(c') > ϵ/(N − 1))
I3        return False
I4    return True

D0 fun dynamicCheck(buff)
D1   return InvSuff(buff) ∧ LetPRComm(buff)

N0 indication (rb, deliver(n, buff(buff)))
N1   if (self ≠ n)
N2     exec(buff)
N3     issue request (pl, send(n, ack(buff)))

A0 indication (pl, deliver(n, ack(c)))
A1   p ← p[n ↦ (p(n) \ {c})]
A2   foreach (c ∈ wq) issue request(call(c))
A3   wq ← ∅

E0 fun exec(c)
E1   σ ← update(c)(σ);  v ← retv(c)(σ)
E2   issue indication ret(c, v)

X0 indication(bro, ret(c, v))
X1   issue request (pl, send(orig(c), ack(c)))
X2   issue indication ret(c, v)
```

Fig. 8: Recency-Aware Protocol

(at E_2). Otherwise, if the bound is violated, the buffer is flushed and broadcast to other replicas (at R_5) and the call c is added to the waiting queue wq (at R_6) for later execution. On the other hand, if the conditions for local execution don't pass, the temporary pending map p' is calculated by updating the pending map p with the new call c (at R_{14}-R_{15}). Also, the bound is checked again to decide whether the call can be executed (R_{18}), or, it is added to the waiting queue wq. If the bound check passes (at R_{16}), the pending map p is updated with the p' (at R_{17}) and the call is sent to the underlying basic replicated object bro for synchronization. (When the return indication for c is received from bro (at X_0), an acknowledgement is issued to the originating replica of c to indicate its successful execution (at X_1), and the indication is issued to the client (at X_2).) However, if the bound is violated by the single call c, the buffer is flushed and the call c is inserted to the waiting queue wq.

Each replica rep maintains a map p that maps each replica rep' to the set of pending calls from rep to rep'. The pending map is updated when a new call c is executed: (1) The call c is sent to the basic replication object bro (at R_{11} and R_{16}). The call c is added to the entry for each replica. (2) The call c is executed locally (at R_{20}). When the call is executed locally, the map is updated with the new buffer $c \cdot buff$.

The protocol keeps a set of correct replicas up, and removes a replica from the set if the prefect failure detector issues a crash event for that replica (at F_1). The function $\mathsf{InBound}(p)$ checks whether the bound in other correct replicas can be violated. If the sum of all the pending calls along with the new call c for a replica exceeds the staleness bound (at I_2), the function returns false.

When a buffer is delivered (at N_0), if the current replica is not the originating replica of the buffer, the buffer is executed (at N_1-N_2). It also sends an acknowledgement message to the originating replica of the buffer using pl (at N_3). Upon receiving an acknowledgement message from a replica n (at A_0), the pending map of the current replica is updated (at A_1) by removing the call from the set of pending calls for n. Also, the calls in the waiting queue wq are requested again (at A_2-A_3).

6 Extra Experiment

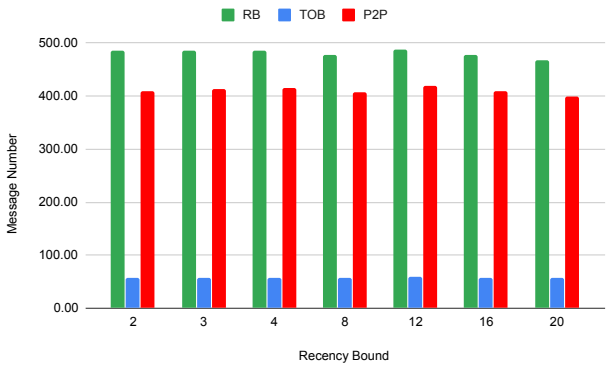
Recency	2	3	4	8	12	16	20
SMT Solver run time (ms)	8.06	7.52	7.98	7.61	7.45	7.34	7.31

Table 1: SMT Solver Run Time for Movie Use-case

Recency	21	30	40	60	80	100	120	140	160	180	200
SMT Solver run time (ms)	7.56	7.54	7.43	9.54	8.97	8.95	8.90	9.06	8.84	8.78	9.03

Table 2: SMT Solver Run Time for Bank Use-case

(a)



(b)

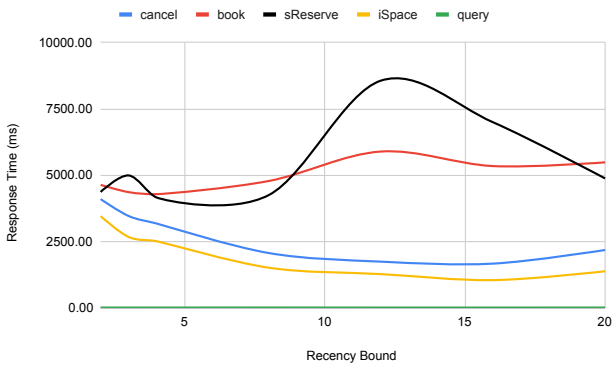
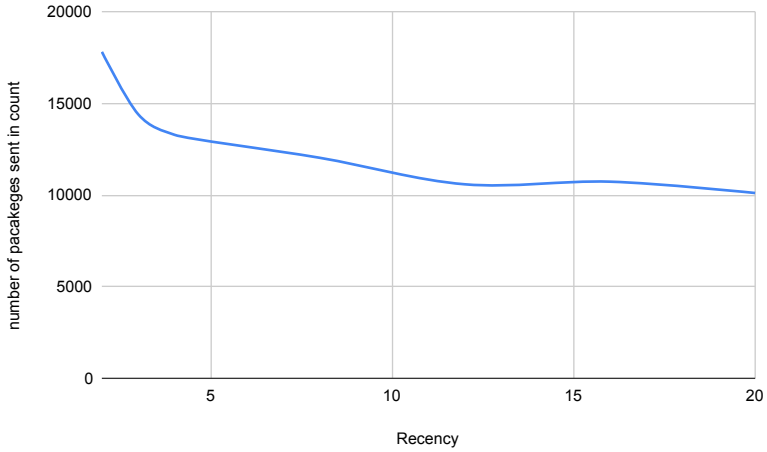


Fig. 9: Effect of recency on coordination load and response time. (a) and (b) show the movie use-case. The frequencies of methods cancel, Book, query, specialReserve, increaseSpace are 20%, 30%, 5%, 20% and 25%.

(a)



(b)

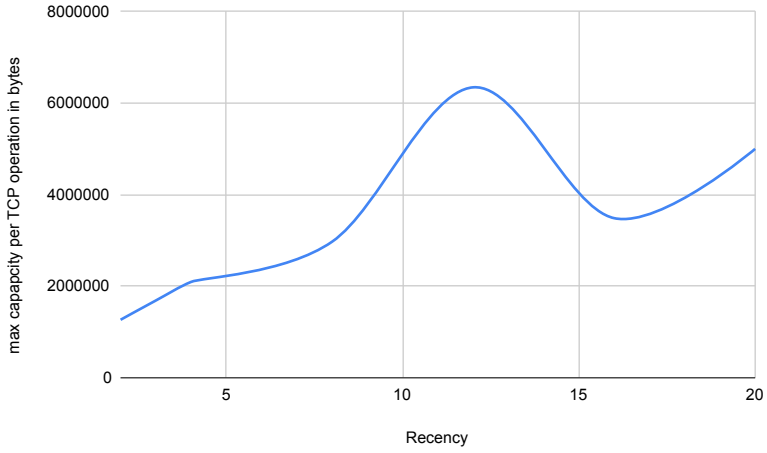


Fig. 10: Effect of recency bound on network.