13 Appendix

13.1 Correctness of Frashokereti: Intuition

Before we begin, we observe that at any point in time, the (current) state of a process is induced by its log, by definition. Thus, when proving the correctness properties, we consider the log of the process.

Theorem 7. *Every ORDT has a proper, non-aborting, optimistic protocol.*
12, 10, 11, and 13 for Frashokereti.

Lemma 6. *Frashokereti preserves integrity.*

For the integrity property, we illustrate the steps of the proof alongside an example in Fig. 6. Whenever a process executes a call, it adds that call to its log. Thus, to show Frashokereti preserves integrity, we consider any log generatable by Frashokereti. The argument is inductive on an incoming call c: after inserting c into the local log, every call is permissible in its (new) pre-state. For this sketch, we only consider when c is a remote call (*i.e.*, works-on($Alice, q_1$) from p_1). The remote handler splits the tentative log of the current process p into two parts, ℓ_1 and ℓ_2, and c is inserted between. We show (1) c is permissible in the post-state of ℓ_1 (called σ_{ℓ_1}), and (2) every call in ℓ_2 is permissible in its new pre-state.

The proof of (1) is done in two steps. First, we derive permissibility for c from its local execution at $home(c)$, that is, p_1. We reconstruct the stable state σ_s^H of p_1 (at the time of executing c) at the current process p, without altering σ_{ℓ_1}, the pre-state of c. To argue this, let H be the set of calls stable for p_1 (*i.e.*, add-project(q_1) and add-employee(Alice)). We show for every call c_h in H, and every call c_p in p but not in H (*i.e.*, add-project(r_2)), if $c_p \prec_{t_p} c_h$, then c_h and c_p state-commute. This allows us to state-commute every call c_h before every c_p call without changing σ_{ℓ_1}. We can re-arrange the calls so that the first $|H|$ calls of p are the calls of H resulting in a state σ_H^p. By convergence, which we prove later, σ_H^p is equal to σ_s^H.

By (LH.1), c must be permissible in this state σ_s^H and thus, in σ_H^p. The second step is to show c $\mathcal{P}_R$-commutes with every call between this state and σ_{ℓ_1}. These calls are one of two types: (t_1) calls causally before c (*i.e.*, add-project(r_2)) or (t_2) calls concurrent to c (*i.e.*, add-employee(Bob)). Calls of type t_1 are in the tentative log of p_1, and by (LH.2), c must $\mathcal{P}_R$-commute with them. By (RH.1), every t_2 call preceding σ_{ℓ_1} must be prior-or-incomparable with c. Thus, c is permissible in σ_{ℓ_1}.

To show (2), we repeat a similar argument for each call in ℓ_2. The part ℓ_2 is empty in this simple example.

Next, we work towards proving convergence. The following lemma states that the log of every process always adheres to two invariants. The first invariant says concurrent calls on comparable methods are executed in their $\prec_{\mathcal{R}}$ order. The second invariant says causally related calls on comparable methods are executed in their causal order.

Lemma 7. *For any pair of distinct calls c_1 and c_2 in a log ℓ of a process p,*

$$\left[(c_1 \parallel_{co} c_2 \wedge c_1 \prec_{\mathcal{R}} c_2) \vee (c_1 \rightarrow_{co} c_2 \wedge c_1 \succ_{\mathcal{R}} c_2)\right] \Rightarrow c_1 \prec_{\ell} c_2.$$

The argument is again inductive on an incoming call c. For this sketch, we only consider when c is a remote call, and c^* is the call chosen by the remote handler in (RH.1), as the other cases are relatively straightforward. Recall that c is inserted right before c^*, so the following properties prove the lemma.

(p.1) For all calls a such that $a \rightarrow_{co} c$ and $a \succ_{\mathcal{R}} c$, we have $a \prec_{\ell} c^*$.
(p.2) For all calls b such that $b \parallel_{co} c$ and $b \prec_{\mathcal{R}} c$, we have $b \prec_{\ell} c^*$.
(p.3) For all calls d such that $c \parallel_{co} d$ and $c \prec_{\mathcal{R}} d$, we have $d = c^*$ or $c^* \prec_{\ell} d$.

For each property, we perform case analysis on its causal/concurrency relation with c^*, *e.g.*, to prove (p.1) holds, we consider three cases: $c^* \parallel_{co} a$, $c^* \rightarrow_{co} a$, and $a \rightarrow_{co} c^*$. As an example, consider the case where $c^* \parallel_{co} a$ and one of its interesting subcases: $c \prec_{\mathcal{R}} a$. By (p.1), we know $a \rightarrow_{co} c$. However, since $c \prec_{\mathcal{R}} a$, the (LH.2) condition requires $home(c)$ to first stabilize a before locally executing c. By definition of the stabilizer, a is only stabilized at $home(c)$ after executing all calls concurrent to a. The case assumption is that $c^* \parallel_{co} a$, implying c^* was also executed by $home(c)$, before executing c. Therefore, $c^* \rightarrow_{co} c$, contradicting the (RH.1) condition that $c^* \parallel_{co} c$.

Lemma 8. *Any pair of calls that appear in opposite orders across the logs of two processes have incomparable methods under $\prec_{\mathcal{R}}$.*

Consider the contra-positive statement: any pair of calls that are comparable under $\prec_{\mathcal{R}}$ appear in the same order across processes. Consider two calls c_1 and c_2 that are comparable, *i.e.*, $c_1 \succ\!\prec_{\mathcal{R}} c_2$. These calls are either concurrent or causally related. Without loss of generality, if the former, we can apply the first disjunct of Lemma 7 and if the latter, we can apply the second disjunct of Lemma 7.

Lemma 9. *Frashokereti preserves convergence.*

In Frashokereti, if a process executes a call, then that call appears in its log. Thus, if two processes have executed the same set of calls, then their logs contain exactly the same calls. Convergence then follows directly from Lemma 8 and Theorem 1.

Lemma 10. *Frashokereti satisfies propagation.*

Proof. If a correct process accepts a call, it sends that call to every other process using reliable broadcast. Reliable broadcast directly ensures that every correct process eventually receives that call. Since a process executes every remote call it receives, propagation follows. □

Lemma 11. *Frashokereti is non-aborting.*

We prove that every correct process p eventually stabilizes and thus commits every call in its history. Recall that for p to stabilize some call c, it must receive from every (remaining) correct process a vector clock larger than c. Recall that our failure detector ensures that p eventually suspects every crashed process. By propagation, every correct process executes every call in the history of p, up to and including c. Afterwards, every correct process eventually executes a new local request (or proxy call) and sends it to p; the vector clock of this call is necessarily larger than that of c.

Lemma 12. *Frashokereti satisfies acceptance.*

By non-abortion, eventually, the stable state catches up to the current state. At this point, if the the call is requested again, it will satisfy the conditions of the local handler, and thus accepted.

Lemma 13. *Frashokereti is optimistic.*

By inspection of the local handler of Frashokereti, we see that when a process receives a request, the process immediately issues either a not-accept or tentative-return response based only on its local state.

Proof. By inspection of our handlers, we can verify Frashokereti is optimistic. For a local call, a process decides whether to execute the call based solely on its stable state, the calls in its tentative log, and $\prec_{\mathcal{R}}$. Each of these are locally accessible variables and do not require messages from any other processes. For a remote call, a process decides where to the call based on the calls in its tentative log and $\prec_{\mathcal{R}}$, which again, are locally accessible. □

13.2 Correctness of Frashokereti: Proofs

Lemma 14. *For any pair of distinct calls c_1 and c_2 in a log ℓ of a process p,*

$$\left[(c_1 \parallel_{co} c_2 \wedge c_1 \prec_{\mathcal{R}} c_2) \vee (c_1 \rightarrow_{co} c_2 \wedge c_1 \succ_{\mathcal{R}} c_2)\right] \Rightarrow c_1 \prec_{\ell} c_2.$$

Proof. The argument is inductive. Let Q be the following predicate over logs ℓ:

$$Q(\ell) \iff \forall c_1, c_2 \in \ell.\ \left[c_1 \neq c_2 \wedge \left[(c_1 \rightarrow_{co} c_2 \wedge c_1 \succ_{\mathcal{R}} c_2) \vee (c_1 \parallel_{co} c_2 \wedge c_1 \prec_{\mathcal{R}} c_2)\right]\right] \Rightarrow c_1 \prec_{\ell} c_2.$$

Consider some log ℓ of process p. In the base case, ℓ is empty and $Q(\ell)$ trivially holds. For the inductive hypothesis, assume $Q(\ell)$ holds for an arbitrary log constructable by Frashokereti. In the inductive step, p receives and executes a call c. Let ℓ' be the new log of p that includes c. We show $Q(\ell')$ holds.

It is straightforward that if c is a local call, then $Q(\ell')$ holds: the local handler adds c to the end of the log without changing the order of any calls already in the log. Additionally, since c is causally after and executed after every call in ℓ, it follows that $Q(\ell')$.

Next, consider when c is a remote call. We focus on c^*, the call chosen by the remote handler in (RH.1). We first verify that if c^* does not exist, *i.e.*, c is executed at the end of the tentative log, $Q(\ell')$ holds. By causal delivery, calls causally before c have already been received and executed. Since c is executed at the end of the tentative log, c comes after every call it is causally after in the log, affirming the first disjunctive implication. The second disjunct is a direct implication of the condition in (RH.1); if there is a call concurrent to c and succeeds it in $\prec_{\mathcal{R}}$, then that call would be chosen as c^*. Since c^* does not exist, no such call exists within the log and thus $Q(\ell')$ holds.

Moving forward, we assume c^* exists. We show the following three properties.

(p.1) For all calls a such that $a \rightarrow_{co} c$ and $a \succ_{\mathcal{R}} c$, we have $a \prec_{\ell} c^*$.
(p.2) For all calls b such that $b \parallel_{co} c$ and $b \prec_{\mathcal{R}} c$, we have $b \prec_{\ell} c^*$.
(p.3) For all calls d such that $c \parallel_{co} d$ and $c \prec_{\mathcal{R}} d$, we have $d = c^*$ or $c^* \prec_{\ell} d$.

These properties imply that when $\overset{\circ}{\ell}$ is split in (RH.2), we have $a, b \in \ell_1$ and $d \in \ell_2$. Since calls of ℓ' are re-executed in order of ℓ_1, c, ℓ_2 in (RH.3), it holds that $Q(\ell')$.

We show each property in turn, beginning with (p.1). Consider some call a such that $a \rightarrow_{co} c$ and $a \succ_{\mathcal{R}} c$. There are three cases, depending on the causal/concurrent relation between a and c^*: (a1) $c^* \parallel_{co} a$, (a2) $c^* \rightarrow_{co} a$, or (a3) $a \rightarrow_{co} c^*$.

Case a1. $c^* \parallel_{co} a$ — By (p.1), we know $a \succ_{\mathcal{R}} c$, giving us two subcases.
Subcase 1. $a \prec_{\mathcal{R}} c$ — By (RH.1), we know $c \prec_{\mathcal{R}} c^*$; thus, transitively $a \prec_{\mathcal{R}} c^*$. By the inductive hypothesis, we get $a \prec_{\ell} c^*$.
Subcase 2. $c \prec_{\mathcal{R}} a$ — We show this subcase cannot happen, by contradiction. By (p.1), we have $a \rightarrow_{co} c$, meaning $a \in \ell_{home(c)}$ when $home(c)$ executes c. However, since $c \prec_{\mathcal{R}} a$, $home(c)$ can only execute c after first stabilizing a, due to the (LH.2) condition. By definition of the stabilizer, $home(c)$ can only stabilize a after executing all calls concurrent to a. This includes c^*, by (a1). Thus, $c^* \in \ell_{home(c)}$ when $home(c)$ executes c, implying $c^* \rightarrow_{co} c$, contradicting the condition of (RH.1) that c and c^* are concurrent.

Case a2. $c^* \rightarrow_{co} a$ — We show this case also cannot happen, by contradiction. Since $a \rightarrow_{co} c$ by (p.1), transitively, we get $c^* \rightarrow_{co} c$. This contradicts the condition of (RH.1) that $c \parallel_{co} c^*$.

Case a3. $a \rightarrow_{co} c^*$ — Similar to case (a1), we have two subcases based on $a \succ_{\mathcal{R}} c$.
Subcase 1. $a \prec_{\mathcal{R}} c$ — By (RH.1), we know $c \prec_{\mathcal{R}} c^*$; thus, transitively $a \prec_{\mathcal{R}} c^*$ and further, $a \succ_{\mathcal{R}} c^*$. By the inductive hypothesis, we get $a \prec_{\ell} c^*$.
Subcase 2. $c \prec_{\mathcal{R}} a$ — We show this subcase cannot happen by focusing on the execution of c^* itself. Let ℓ^* be the log of p right after executing c^*. Since $a \rightarrow_{co} c^*$, it must be that $a \in \ell^*$.

Towards contradiction, suppose $c^* \prec_{\ell^*} a$. Clearly, c^* cannot be local to p as local calls are always executed at the end of the log. Thus, it remains to consider when c^* is a remote call. In the remote handling of c^*, (RH.1) chooses a call c^2 such that $c^* \parallel_{co} c^2$, $c^* \prec_{\mathcal{R}} c^2$, and $c^2 \prec_{\ell^*} a$. In fact, this implies a chain of calls $c^* - c^2 - c^3 - \ldots - c^k$, where $c^i \parallel_{co} c^{i+1} \wedge c^i \prec_{\mathcal{R}} c^{i+1} \wedge c^i \prec_{\ell^*} c^{i+1}$, for $i \in \{2, \ldots, k\}$, and $c^k \prec_{\ell^*} a$. Let this chain be maximal in the sense that there does not exist a call between the last call of the chain and a such that the last call is concurrent to it and precedes it under $\prec_{\mathcal{R}}$. We claim such a chain cannot exist by arguing that c^k, the last element of the chain, cannot exist. This gives us three further subsubcases, for which we show each leads to contradiction.

(*Subsubcase 2.1.*) $c^k \parallel_{co} a$ — Consider $home(c)$ at the time of executing c. By (p.1), we know $a \in \ell_{home(c)}$. Further, since $c \prec_{\mathcal{R}} a$ by the subcase assumption, for $home(c)$ to execute c, it must first stabilize a. This means $home(c)$ must first execute all calls concurrent to a, *i.e.*, $c^k \in \ell_{home(c)}$ as well. However, since $c \prec_{\mathcal{R}} c^*$ by (RH.1) and $c^* \prec_{\mathcal{R}} \ldots \prec_{\mathcal{R}} c^k$ by definition of the chain, c cannot be executed until c^k is also stable, by the (LH.2) condition. We can repeat this reasoning for the entire chain: every call in the chain is stable in $\ell_{home(c)}$ before executing c, notably c^*. However, this implies $c^* \rightarrow_{co} c$, contradicting the concurrency condition of (RH.1).

(*Subsubcase 2.2.*) $c^k \rightarrow_{co} a$ — This time, consider $home(c^*)$ at the time of executing c^*. By (a3), $home(c^*)$ executes c^* when $a \in \ell_{home(c^*)}$. By the subsubcase assumption, transitively, we get $c^k \in \ell_{home(c^*)}$ when $home(c^*)$ executes c^*. By definition of the chain, we know $c^* \prec_{\mathcal{R}} \ldots \prec_{\mathcal{R}} c^k$. Thus, for $home(c^*)$ to execute c^*, it must first stabilize c^k. Similar to the previous subsubcase, to stabilize c^k, it must first execute and stabilize c^{k-1}, and repeatedly, for the entire chain, all the way to c^2. This contradicts the concurrency condition of (RH.1) between c^* and c^2.

(*Subsubcase 2.3.*) $a \rightarrow_{co} c^k$ — Consider p. If this were true, then during the execution of c^k, a is already in the log. For c^k to precede a, it must be that c^k is remote. Further, (RH.1) chooses a call c^{k+1} such that $c^k \parallel_{co} c^{k+1} \wedge c^k \prec_{\mathcal{R}} c^{k+1}$. However, since the chain is assumed to be maximal, no such c^{k+1} can exist, a contradiction.

Now that we have shown the last element of this chain, c^k, cannot exist, the remainder of the proof is straightforward. Since the chain must be finite, and a last element c^k cannot exist, then the entire chain cannot exist. This means such a c^2 cannot exist either. If c^2 does not exist, then (RH.1) on c^* must choose a call that succeeds a in ℓ^*. Therefore, $a \prec_{\ell^*} c^*$. Since the order between executed calls never changes, it holds that $a \prec_{\ell} c^*$ as well.

Next, consider property (p.2). Similar to the proof of (p.1), there are three cases depending if (b1) $c^* \parallel_{co} b$, (b2) $c^* \rightarrow_{co} b$, or (b3) $b \rightarrow_{co} c^*$, though much simpler. We first observe that $b \prec_{\mathcal{R}} c^*$. This is because, by (p.2), we have $b \prec_{\mathcal{R}} c$. Then by (RH.1), we know $c \prec_{\mathcal{R}} c^*$, thus transitively, $b \prec_{\mathcal{R}} c^*$.

Case b1. $b \parallel_{co} c^*$ — Together with the observation, we can apply the induction hypothesis, implying $b \prec_{\ell} c^*$.

Case b2. $c^* \rightarrow_{co} b$ — We show this case cannot happen, by contradiction. Then $c^* \in \ell_{home(b)}$ when $home(b)$ executes b. By our observation, $b \prec_{\mathcal{R}} c^*$. Using the (LH.2) condition on b, $home(b)$ can execute b only after stabilizing c^* first. By (RH.1), c and c^* are concurrent. Thus, to stabilize c^*, $home(b)$ must first execute c as well. This implies $c \rightarrow_{co} b$, contradicting (p.2) that $c \parallel_{co} b$.

Case b3. $b \rightarrow_{co} c^*$ — The observation implies $b \succ_{\mathcal{R}} c^*$. By the inductive hypothesis, we get $b \prec_{\ell} c^*$.

Lastly, we show property (p.3). It is straightforward that such a call d cannot precede c^* in ℓ, for if it did, the remote handler for c would have chosen d instead of c^* in (RH.1). More specifically, such a d contradicts the condition that c^* is the first such a call in $\overset{\circ}{\ell}$. □

Lemma 15. *Any pair of calls that appear in opposite orders across the logs of two processes have incomparable methods under* $\prec_{\mathcal{R}}$.

Proof. We show the contra-positive statement: any pair of calls that are comparable under $\prec_{\mathcal{R}}$ appear in the same order across processes. Consider two calls c_1 and c_2 that are comparable, *i.e.*, $c_1 \succ\!\prec_{\mathcal{R}} c_2$, and are in the logs ℓ_1 and ℓ_2 of two processes. There are two cases, based on whether the calls are causally ordered or not. First, assume they are causally ordered, without loss of generality: $c_1 \rightarrow_{co} c_2$. By Lemma 14, we have that $c_1 \prec_{\ell_i} c_2$ for $i \in \{1, 2\}$. Next, assume c_1 and c_2 are concurrent, *i.e.*, $c_1 \parallel_{co} c_2$. Since they are comparable, without loss of generality, assume $c_1 \prec_{\mathcal{R}} c_2$. Again by Lemma 14, we have that $c_1 \prec_{\ell_i} c_2$ for $i \in \{1, 2\}$. □

Lemma 16. *Frashokereti preserves convergence.*

Proof. In Frashokereti, if a process executes a call, then that call appears in its log. Thus, if two processes have executed the same set of calls, then their logs contain exactly the same calls. Convergence then follows directly from Lemma 15 and Theorem 1. □

Lemma 17. *Frashokereti preserves integrity.*

Proof. Consider the execution history h and its recent history r_h generated by Frashokereti at some process p. We must show that the invariant $\mathcal{I}$ holds for the resulting state of executing the sequence of calls induced by r_h. As the initial state is assumed to have integrity, in order to show that the pre-state of every call has integrity, we show the post-state of every call has integrity, *i.e.*, every call in the sequence is permissible. We do so inductively, similar to the proof of Lemma 14. In Frashokereti, whenever a process issues a tentative-return response for a call c, it first adds c to its log. Thus, the log ℓ of p reflects the recent history r_h of p: if a response for a call is in r_h, then that call is also in ℓ; if the response for a call c precedes the response for a call c' in r_h, then c precedes c' in ℓ. Thus, to simplify the proof, we can instead argue about the log of p, rather than its recent history.

Define a predicate Q over logs ℓ such that $Q(\ell)$ holds iff every call in ℓ is permissible in its pre-state. In the base case, ℓ_p is empty and $Q(\ell_p)$ trivially holds. For the inductive hypothesis, we assume $Q(\ell_p)$ holds for an arbitrary log constructable by Frashokereti. In the inductive step, process p receives and executes a call c. We show that $Q(\ell'_p)$ holds, where ℓ'_p is the new log of p that includes c.

The local case is simple; the call c is executed at the latest state σ_n. Therefore, the preceding calls stay permissible. It suffices to show c is permissible in σ_n. By (LH.1), c is permissible in the stable state σ_s. By (LH.2), c $\mathcal{P}_R$-commutes with every call between σ_s and σ_n. Therefore, c is permissible in σ_n and $Q(\ell'_p)$ holds.

Next, consider when c is a remote call. The remote handler divides the tentative log $\overset{\circ}{\ell}$ into two parts, ℓ_1 and ℓ_2, then re-executes the calls in order of ℓ_1, c, then ℓ_2 on σ_s. Since calls in ℓ_1 do not move, they are trivially permissible in ℓ'_p. It remains to show c and all calls in ℓ_2 are permissible in their (new) pre-states.

We begin by showing c is permissible in its pre-state, which is the post-state of the ℓ_1 sequence on σ_s. Call this state σ_{ℓ_1}. To do this, we derive permissibility from the log of $home(c)$ when c was locally executed. First, (LH.1) implies c is permissible in the stable state of $home(c)$, which we call σ_s^H. Let H be the set of calls executed in $home(c)$ that resulted in σ_s^H, *i.e.*, the set of stable calls.

The idea is to apply state-commutativity to reorder calls in ℓ_p to reconstruct σ_s^H at p, without changing σ_{ℓ_1}. By causal delivery, every call in H must also be in ℓ_p. We claim that for every call $c_h \in H$, and every call c_p where $c_p \in \ell_p$ and $c_p \notin H$, if $c_p \prec_\ell c_h$, then c_h state-commutes c_p. This claim says every c_h call can be brought before every call that is in ℓ_p but not in H, without changing

σ_{ℓ_1}. There are two cases. In the first case, consider if c_p has been executed by $home(c)$. By Lemma 15, since c_p and c_h are executed in opposite orders across p and $home(c)$, they state-commute. In the second case, consider if c_p has not been executed by $home(c)$. Since c_h is executed before σ_s^H, it is stabilized at $home(c)$; therefore, $home(c)$ must have executed all calls causally before and concurrent to c_h. Thus, if c_p has not been executed by $home(c)$, it must be because $c_h \rightarrow_{co} c_p$. However, since $c_p \prec_\ell c_h$, by Lemma 14, we know $c_p \leftrightarrow_{\mathcal{R}} c_h$, meaning they state-commute.

Using the above state-commutativity claim, we can state-commute every call $c_h \in H$ to the beginning of ℓ, so that the first $|H|$ calls of ℓ are exactly H. Importantly, this does not alter σ_{ℓ_1}. Letting σ_H^p be the post-state of the first $|H|$ calls, we know σ_H^p and σ_s^H are equivalent states by Lemma 16. Since c is permissible in σ_s^H by (LH.1), c is permissible in σ_H^p.

The next step is to show that c $\mathcal{P}_R$-commutes with every call between σ_H^p and σ_{ℓ_1}. We can classify these calls into one of two types: (t.1) calls causally before c and (t.2) calls concurrent with c. We consider each in turn. A call t_1 of type (t.1) is in the tentative log of $home(c)$. This is because if a call is causally before c, it must have been executed by $home(c)$. Additionally, t_1 cannot be stable; otherwise, it would be included in H. Then by (LH.2), we know c $\mathcal{P}_R$-commutes with t_1. A call t_2 of type (t.2) has not been executed in $home(c)$ because it is concurrent with c. But t_2 is in $\overset{\circ}{\ell_p}$, the tentative log of p. It must be that $t_2 \not\leftrightarrow_{\mathcal{R}} c$, otherwise (RH.1) would have chosen t_2 to be c^* in the remote execution of c. This implies c $\mathcal{P}_R$-commutes with t_2 as well. Therefore, c is permissible in σ_{ℓ_1}.

The last task of the proof is to show that every call in ℓ_2 is permissible in its new pre-state in ℓ'. We reapply the argument we made for c, to each $c' \in \ell_2$. That is, reconstruct a state in ℓ' equivalent to the stable state of $home(c')$ during the local execution of c' using the same procedure (regardless of whether c' is local or remote to p). By (LH.1), c' is permissible in this pre-state. Then, we argue c' $\mathcal{P}_R$-commutes with every call between this state and its eventual pre-state in a similar way: classify these calls into one of three types. The first type of calls are analogous to (t.1) calls, *i.e.*, calls that are causally before c'. These have been executed in $home(c')$, thus by (LH.2), c' $\mathcal{P}_R$-commutes with each of these calls. The second type of calls are analogous to (t.2) calls, *i.e.*, calls that are concurrent with c'. If c' $\mathcal{P}_R$-conflicts with a concurrent call, by Lemma 14, c' precedes that call in ℓ'. Therefore, calls that are concurrent with and precede c' in ℓ' are prior-or-incomparable with c', *i.e.*, c' $\mathcal{P}_R$-commutes with each of these calls. The third type of calls are those causally after c'. For a call t_3, if c' is causally before t_3 and c' $\mathcal{P}_R$-conflicts with t_3, then by Lemma 14, c' precedes t_3 in ℓ'. Therefore, for any call that is causally after c' and precedes it in ℓ', c' $\mathcal{P}_R$-commutes with each of those calls. □

Lemma 18. *Frashokereti satisfies propagation.*

Proof. Reliable broadcast directly ensures that if a correct process executes a call, then every correct process will receive and thus execute that call. □

Lemma 19. *Frashokereti is non-aborting.*

Proof. By Lemma 18, if a correct process executes a call, then every process will execute that call. Then it remains to argue that every call is eventually stabilized by every correct process, as stability implies commitment. Consider some call c at a correct process p. Recall that for p to stabilize c, it must receive from every (remaining) correct process a vector clock larger than c. By the failure detector, p eventually suspects every crashed process. By reliable broadcast, every correct process eventually executes every call in the history of p, up to and including c. That is, every correct process has a local vector clock at least that of c. After, every correct process eventually executes a new local request (or a proxy call if it does not receive a new request by the timeout), and sends it

to p. The attached vector clock of this call is necessarily larger than c. Therefore, p receives a vector clock larger than c from every correct process, and c is stabilized. Lastly, when a call is stabilized, it is added to the stable state. By simple inspection of our handlers, no call is ever inserted before the stable state, implying c is committed. □

Lemma 20. *Frashokereti satisfies acceptance.*

Proof. Suppose the client requests a call permissible at the current state of some correct process infinitely many times. Eventually, if no additional requests are made to the system, by Lemma 19, the stable state is eventually the current state. At this point, if the client requests the call again, the conditions of the local handler are satisfied, prompting the process to execute the call. □

Lemma 21. *Frashokereti is optimistic.*

Proof. By inspection of the local handler of Frashokereti, we can see that when a process receives a request, the process immediately issues either a not-accept or tentative-return response based only on its local state. □

Lemma 22. *Frashokereti is response-matched.*

Proof. We can verify that a process issues a response for a call only if some process previously received a request for that call by directly examining the handlers. There are three types of responses: not-accept, tentative-return, and commit. A process only ever responds not-accept within its local handler. The local handler of a process only executes upon receiving a request. A process can issue a tentative-return response for a call in (a) its local handler and (b) when it re-executes the call in the remote handler. The former case is the same as before. In the latter case, the remote handler of a process only executes upon receiving a message (call) from another process. A process only broadcasts a message (call) upon executing that call within its own local handler. Thus, case (b) reduces to some other process satisfying case (a). Lastly, a process issues a commit response for a call after stabilizing that call in its log. A process adds a call to its log only after issuing a tentative-return response for that call, which we already showed implies that some process received a request for that call. □

Lemma 23. *Frashokereti is commit-final.*

Proof. A process only issues a commit response for a call it previously issued a tentative-return response for. Since a call (request) can only be processed by a local handler once, this implies that if a process were to issue a tentative-return response after a commit response for a call, it must be through the remote handler. A process can only issue a tentative-return response through its remote handler for calls in its tentative log. However, when a process issues a commit response for a call, that call is removed from its tentative log (and can never return). Therefore, a process cannot issue a tentative-return response for a call after issuing commit for it. □

Lemma 24. *Frashokereti is commit-unique.*

Proof. A process issues a commit response for a call upon stabilizing it. A process stabilizes the first call in its log, after which the call is removed and never returns. Therefore, at most one commit response is issued for each call. □

Lemma 25. *Frashokereti complies with every ORDT.*

Proof. We should prove that the sequence of complete calls induced by the recent history of every process is in the sequential specification of O. The result is immediate from Lemma 31, Lemma 30 and Lemma 29. Lemma 31 splits the recent history into the committed part and the

uncommitted part. Then Lemma 31 shows that the committed part complies with the sequential specification and results in the stable state σ_s. Lemma 30 shows that the sequence of complete calls that the sequential execution of the tentative log on the stable state σ_s generates is the uncommitted part. Therefore, the uncommitted part complies with the sequential specification as well. □

The following lemmas draw a correspondence between the sequence of complete calls generated by executing the log on the initial state σ_0 and the sequence of complete calls induced by the recent history of the process. Specifically, the two sequences are equal. This serves two purposes. First, this will directly prove that Frashokereti complies with every ORDT. Second, this allows us to reason about the logs of processes, rather than the requests and responses in their histories.

For the following lemmas, let h be the history and h_r the recent history of a process p. The *recent committed history* h_r^c of p is the projection of h_r over the committed calls in h. The *recent uncommitted history* h_r^u of p is the projection of h_r over the calls that have received a tentative-return response, but not a commit response in h.

Lemma 26. *The sequence of complete calls generated by executing the tentative log $\overleftarrow{\ell}$ starting from the stable state σ_s is equal to the sequence of complete calls induced by the recent uncommitted history.*

Proof. Let $\overleftarrow{\ell}$ be the tentative log and h_r^u be the recent uncommitted history of a process p. Denote the sequence of complete calls generated by executing $\overleftarrow{\ell}$ on the stable state σ_s of p as $s(\overleftarrow{\ell})$. Denote the sequence of complete calls induced by h_r^u as $s(h_r^u)$.

The argument is inductive on the number of events that have been handled by p. We argue that $s(\overleftarrow{\ell}) = s(h_r^u)$.

Base case: Since p has handled zero events, the tentative log $\overleftarrow{\ell}$ is empty and p has not issued tentative-return for any calls. That is, h_r^u is empty as well. Therefore, $s(\overleftarrow{\ell}) = s(h_r^u)$ as they are both the empty sequence.

Inductive hypothesis: Assume after handling k events, $s(\overleftarrow{\ell})_k = s(h_r^u)_k$.

Inductive step: We must show that after p handles $k+1$ events, the invariant holds, *i.e.*, $s(\overleftarrow{\ell})_{k+1} = s(h_r^u)_{k+1}$. We consider three cases, corresponding to when the tentative log $\overleftarrow{\ell}$ is changed by the local handler, remote handler, and stabilizer. We elide events where the tentative log does not change.

Case 1: *Local handler adds call c to $\overleftarrow{\ell}$.* When the local handler adds a call c to $\overleftarrow{\ell}$, c is always added to the end. After, p issues tentative-return$(c : v)$ for the call (and issues no other responses). Thus, tentative-return$(c : v)$ appears at the end of h_r^u. Therefore, $s(\overleftarrow{\ell})_{k+1} = s(h_r^u)_{k+1}$.

Case 2: *Remote handler inserts call c into $\overleftarrow{\ell}$ before index i.* The value v of the tentative-return response that p issues for c is calculated from the pre-state in which it is inserted, *i.e.*, σ_{i-1}. Similarly, p recalculates the return values of all calls at index i and greater within the tentative log and issues a new tentative-return response for them. In other words, Frashokereti re-executes all calls in the suffix ℓ_2 and issues a new tentative-return response for each re-executed call after computing the new return value with respect to the new pre-state of the call. These tentative-return events are then, in order, the latest in the history of p, and thus the latest in h_r^u. Therefore, the $s(\overleftarrow{\ell})_{k+1} = s(h_r^u)_{k+1}$.

Case 3: *O response Yes.* When the stabilizer O responds Yes, the first call in $\overleftarrow{\ell}$ is removed *i.e.*, $\overleftarrow{\ell}\ [0]$. A commit response is issued for this call; thus it is removed from h_r^u as well. By the induction hypothesis, since $s(\overleftarrow{\ell})_k = s(h_r^u)_k$, that means the first element of $s(\overleftarrow{\ell})_k$ is the same as the first element of $s(h_r^u)_k$. Thus, the first elements of $\overleftarrow{\ell}$ and h_r^u are removed. Further, the new stable state σ_s' is the result of applying $\overleftarrow{\ell}\ [0]$ to σ_s. □

Lemma 27. *Starting from σ_0, the execution of the complete calls induced by the recent committed history of every process is in the sequential specification of O, and result in the stable state σ_s.*

Proof. Let h_r^c denote the recent committed history of a process p. Denote $s(h_r^c)$ as the sequence of complete calls induced by h_r^c. We argue that $s(h_r^c) \in \mathsf{S}$, where S is the sequential specification of O. The argument is inductive on the number of events handled by p.

Base case: Since p has handled zero events, the recent committed history h_r^c is empty. Thus, $s(h_r^c) = \varnothing$. The empty sequence is in S, and $\sigma_0 = \sigma$.

Inductive hypothesis: Assume after handling k events, $s(h_r^c)_k \in \mathsf{S}$. Further, the stable state σ_s is post-state of the last complete call in $s(h_r^c)_k$.

Inductive step: Consider when the stabilizer O responds Yes for the first call $c_1 = \overset{\circ}{\ell}\,[0]$ in the tentative log of p. First, p issues commit(c_1), meaning c_1 is added to the end of the recent committed history h_r^c. Then, $s(h_r^c)_{k+1}$ is equal to $s(h_r^c)$ with $\langle c_1 : v_1 \rangle$ appended to the end. By Lemma 29, applying c_1 to σ_s results in the same return value v_1 as the last tentative-return response for c_1. Since $\langle \sigma_s', v_1 \rangle = c_1(\sigma_s)$, by the inductive hypothesis, $s(h_r^c)_{k+1} \in \mathsf{S}$ and results in the new stable state σ_s'. □

Lemma 28. *In every recent history, all calls in the recent committed history precede all calls in the recent uncommitted history.*

Proof. Let h_r^c denote the recent committed history and h_r^u denote the recent uncommitted history. We argue all calls in h_r^c precede all calls in h_r^u. The argument is inductive on the number of events handled by process p.

Base case: Since p has handled zero events, the recent history is empty. Thus the invariant is vacuously true.

Inductive hypothesis: Assume after handling k events, all calls in $(h_r^c)_k$ precede all calls in $(h_r^u)_k$.

Inductive step: We must show after handling $k+1$ events, all calls in $(h_r^c)_{k+1}$ precede all calls in $(h_r^u)_{k+1}$.

We consider three cases corresponding to when the recent history changes, corresponding to the local handler, remote handler, and stabilizer. We elide events where the recent history does not change.

Case 1: *Local handler adds call c to $\overset{\circ}{\ell}$.* A tentative-return response is issued for a new call c. This response is added to the end of $(h_r^u)_k$, thereby the inductive hypothesis, all calls in $(h_r^c)_{k+1}$ precede all calls in $(h_r^u)_{k+1}$.

Case 2: *Remote handler adds call c to $\overset{\circ}{\ell}$ at index i.* By Lemma 29, $\overset{\circ}{\ell}$ contains only uncommitted calls. Therefore, when p inserts c at index i, p issues tentative-return for c and only re-issues calls in $(h_r^u)_k$, specifically the calls in ℓ_2. Thus by the inductive hypothesis, all calls in $(h_r^c)_{k+1}$ precede all calls in $(h_r^u)_{k+1}$.

Case 3: *O responds Yes.* A commit response is issued for $c_1 = \overset{\circ}{\ell}\,[0]$, the first element of $\overset{\circ}{\ell}$. Consider the state before the handler is executed. By Lemma 29, the tentative-return response of c_1 is the first in $(h_r^u)_k$. After the handler is executed, c_1 moves from the recent uncommitted history, to the recent committed history. In other words, $c_1 \notin (h_r^u)_{k+1}$ and $c_1 \in (h_r^c)_{k+1}$. □

For the following lemmas, let h be the history and h_r be the recent history of a process p. The *recent committed history* of p is the projection of h_r over the committed calls of h. The *recent uncommitted history* of p is the projection of h_r over the uncommitted calls of h.

Lemma 29. *The sequence of complete calls generated by the execution of the tentative log $\overset{\circ}{\overline{\ell}}$ starting from the stable state σ_s is equal to the sequence of complete calls induced by the recent uncommitted history.*

Proof. The proof is by induction on the handler steps of the protocol.
Base:
The tentative log $\overset{\circ}{\overline{\ell}}$ and the recent history are both empty. Trivial.
Inductive:
Consider the tentative log $\overset{\circ}{\overline{\ell}}$, and the sequence s of complete calls induced by the recent uncommitted history.

$\overset{\circ}{\overline{\ell}} = t_4, \quad t_5, \quad .., \quad t_6$
$\sigma_s \quad \sigma'_s$
$s = t_4 : v_4, \quad t_5 : v_5, \quad ..., \quad t_6 : v_6$ Induced by recent uncommitted history

We consider two case where the tentative log $\overset{\circ}{\overline{\ell}}$ is changed.

Case (1) In the handler O **response** Yes, a call is removed from $\overset{\circ}{\overline{\ell}}$. The call that is removed is the first element $\overset{\circ}{\overline{\ell}}\ [0] = t_4$. A commit response is issued for this call; thus, it is removed from the recent uncommitted history s as well. Thus, the first elements are removed from both $\overset{\circ}{\overline{\ell}}$ and the recent uncommitted history. Further the new stable state σ' is the result of applying $\overset{\circ}{\overline{\ell}}\ [0] = t_4$ to σ_s. Thus, the relation between the new $\overset{\circ}{\overline{\ell}}$ and the new recent uncommitted history is immediate from the induction hypothesis.

Case (2): In the remote handler, a call t' is inserted in the middle of the tentative log $\overset{\circ}{\overline{\ell}}$.

$\overset{\circ}{\overline{\ell}} = t_4, \quad t_5, \quad t_6$ Tentative log
σ_s
$s = t_4 : v_4, \quad t_5 : v_5, \quad t_6 : v_6$ Induced by the recent uncommitted history
New state
$\overset{\circ}{\overline{\ell}}' = t_4, \quad t_5, \quad t', \quad t_6$ Tentative log
σ_s
$s' = t_4 : v_4, \quad t_5 : v_5, \quad t' : v', \quad t_6 : v'_6$ Induced by the recent uncommitted history

The value v' of the tentative-return that the protocol issues for the new call t is calculated in the post-state of t_5. Similarly, the protocol recalculates the new return value of t_6 based on its new pre-state, and issues a new tentative-return response for it. These tentative-return events are the latest tentative-return events for t' and t_6 in the history, and appear in the new recent uncommitted history. Therefore, the execution of the new tentative log $\overset{\circ}{\overline{\ell}}'$ on the stable state σ_s results in the sequence of complete calls induced by the new recent uncommitted history s'. □

Lemma 30. *Starting from σ_0, the execution of the complete calls induced by the recent committed history of every process is in the sequential specification of O, and result in the stable state σ_s.*

Proof. The proof is by induction on the handler steps of the protocol.

Base: The recent committed history is empty. An empty sequence is in the sequential specification, and $\sigma_0 = \sigma_s$

Inductive case:

Consider the recent committed history.

$s = c_1 : v_1, \quad c_2 : v_2, \quad .., \quad c_3 : v_3$ Induced by recent committed history

σ_0 σ_s

By induction hypothesis, we have that $s \in SS$ the sequential specification, and $s(\sigma_0) = (\sigma_s, v_3)$.

In the handler O **response** Yes, when the protocol issues a commit response for the call $\overset{\circ}{\ell}\ [0]$, it further takes the first element of the tentative log $\overset{\circ}{\ell}\ [0]$, and applies it to the stable state σ_s to get the new stable state σ'_s. By Lemma 29, applying $\overset{\circ}{\ell}\ [0]$ to σ_s results in the same return value v as the last tentative-return response of $\overset{\circ}{\ell}\ [0]$. After $\overset{\circ}{\ell}\ [0]$ is committed. Let us consider the new sequence of complete calls induced by recent committed history:

$s = c_1 : v_1, \quad c_2 : v_2, \quad .., \quad c_3 : v_3, \quad \overset{\circ}{\ell}\ [0] : v$

σ_0 σ_s σ'_s

Since $\langle \sigma'_s, v \rangle = \overset{\circ}{\ell}\ [0](\sigma)$, the new sequence s' is in the SS, and results in the new stable state σ'_s. □

Lemma 31. *In the recent history of every process, the committed calls are before the uncommitted calls.*

Proof. The proof is by induction on the handler steps of the protocol.
Base:
The recent history is empty. Trivial.

Inductive:
We consider the steps of three handlers. We assume that before the handler is executed, the committed calls are before the uncommitted calls in the recent history of the process.

(a) Local calls: A tentative-return is issued for a new call. Thus, the uncommitted calls is extended at the end. Therefore, in the recent history, the committed calls stay before the uncommitted calls.

(b) Remote calls: By Lemma 29, $\overset{\circ}{\ell}$ contains only uncommitted calls. The handler issues tentative-return only for calls in $\overset{\circ}{\ell}$. Thus, tentative-return is issued for only uncommitted calls. Thus, similar to the previous case, in the recent history, committed calls stay before uncommitted calls.

(c) The oracle handler: A commit response is issued for $\overset{\circ}{\ell}[0]$, the first element of $\overset{\circ}{\ell}$. Consider the state before the handler is executed. By Lemma 29, the tentative-return response of for the call $\overset{\circ}{\ell}[0]$ is the first in the recent uncommitted history. Now consider the state after the handler is executed. The call $\overset{\circ}{\ell}[0]$ moves from the recent uncommitted history to recent committed history. Therefore, committed calls stay before uncommitted calls. □

13.3 Abstracting and Transforming Relational Schema into ORDTs

- We split add-row into two separate methods add-row-yield and add-row-force.
  - add-row-yield(t, r): add row r to table t, unless there exists another row in t that has the same values as r for a UNIQUE column or PK columns.
  - add-row-force(t, r): add row r to table t. Further, remove any rows in t that have the same values as r for a UNIQUE or PK columns.

- We split delete-row into two separate methods delete-row-cascade and delete-row-no-action.
  - delete-row-cascade(t, ϕ): delete the rows that satisfy the condition ϕ from t. Further, if a deleted row contains a primary key k, then for all tables, delete all rows with a foreign key that refer to k.
  - delete-row-no-action(t, ϕ): delete the rows that satisfy the condition ϕ from t, unless a target row contains a primary key that is referred to by some foreign key.
- We split delete-table into two separate methods drop-table-cascade and drop-table-noaction.
  - drop-table-cascade(t): delete table t. Further, for every row in t with primary key k, delete any row with a foreign key that refers to k from all tables.
  - drop-table-noaction(t): delete table t, unless there exists a row in t containing a primary key that is referred to by some foreign key.

- We split update into six separate methods. We do so incrementally. The presentation order here is different than the main paper. The first is whether or not the update applies to primary keys.
  - update-PK$(t, \phi, \langle \overline{c, v} \rangle)$: if every column $\overline{c}$ is in the primary key of t, for each row r in table t that satisfies condition ϕ, the columns $\overline{c}$ are set to values $\overline{v}$,
  - update-nonPK$(t, \phi, \langle \overline{c, v} \rangle)$: if no column $\overline{c}$ is in the primary key of t, for each row r in table t that satisfies condition ϕ, the columns $\overline{c}$ are set to values $\overline{v}$.
- We then split update-PK by cascade or no-action.
  - update-PK-cascade$(t, \phi, \langle \overline{c, v} \rangle)$: if every column $\overline{c}$ is in the primary key of t, for each row r in table t that satisfies condition ϕ, the columns $\overline{c}$ are set to values $\overline{v}$, Further, for every cell updated, apply the same update to every cell that refers to the updated cell, across all tables.
  - update-PK-no-action$(t, \phi, \langle \overline{c, v} \rangle)$: if every column $\overline{c}$ is in the primary key of t, and no foreign key cell references a cell in a row that satisfies ϕ, for each row r in table t that satisfies condition ϕ, the columns $\overline{c}$ are set to values $\overline{v}$.
- At this point, we have three update methods: update-PK-cascade, update-PK-no-action, and update-nonPK. The last step is to apply yield and force to each of them.
  - update-PK-cascade-force$(t, \phi, \langle \overline{c, v} \rangle)$: if every column $\overline{c}$ is in the primary key of t, for each row r in table t that satisfies condition ϕ, the columns $\overline{c}$ are set to values $\overline{v}$, Further, for every cell updated, apply the same update to every cell that refers to the updated cell, across all tables. Lastly, if r shares the same values with any other rows r' in t across PK columns, delete r' from t.
  - update-PK-cascade-yield$(t, \phi, \langle \overline{c, v} \rangle)$: if every column $\overline{c}$ is in the primary key of t, and if applying the update does not result in two rows with the same values across PK columns, then for each row r in table t that satisfies condition ϕ, the columns $\overline{c}$ are set to values $\overline{v}$, Further, for every cell updated, apply the same update to every cell that refers to the updated cell, across all tables.
  - update-PK-no-action-force if every column $\overline{c}$ is in the primary key of t, and no foreign key cell references a cell in a row that satisfies ϕ, for each row r in table t that satisfies

condition ϕ, the columns $\overline{c}$ are set to values $\overline{v}$. Lastly, if r shares the same values with any other rows r' in t across PK columns, delete r' from t.
  - update-PK-no-action-yield if every column $\overline{c}$ is in the primary key of t, and no foreign key cell references a cell in a row that satisfies ϕ, and if applying the update does not result in two rows with the same values across PK columns, then for each row r in table t that satisfies condition ϕ, the columns $\overline{c}$ are set to values $\overline{v}$.
- The last step is to apply yield and force to update-nonPK to be sensitive of UNIQUE columns.
  - update-nonPK-force if no column $\overline{c}$ is in the primary key of t, for each row r in table t that satisfies condition ϕ, the columns $\overline{c}$ are set to values $\overline{v}$. Lastly, if r shares the same values with any other rows r' in t at a UNIQUE column, delete r' from t.
  - update-yield-nonPK if no column $\overline{c}$ is in the primary key of t, for each row r in table t that satisfies condition ϕ and if applying the update does not result in two rows with the same values across PK columns, for each row r in table t that satisfies condition ϕ, the columns $\overline{c}$ are set to values $\overline{v}$.

13.3.1 Discussion on Transformations. Although the yield and force transformations we discuss can always be applied, they are not always semantically meaningful. That is, although the replicated object can technically repair the invariant with these transformations, they do not reflect the reality of the object. For example, yield and force can be applied to a bank account to transform it into an ORDT. However, this transformation is not semantically meaningful.

If we apply yield to the withdraw method, two concurrent withdraws whose sum exceeds the balance would cause the second withdraw to yield. The two processes would still preserve convergence and integrity, but this is non-consequential as the clients have already received their money. Likewise, if we apply force to withdraw and then set any negative balance to zero, we encounter the same problem. It is non-consequential since the client withdrew money he did not have have. Most would also find this forcing to be semantically unacceptable.

In a bank account, the withdraw method produces external side-effects: the money is given to the client, with no way to address invariant violations since the client can simply walk away. The programmer should not approve the transformation, although it is technically possible..

The inapplicability of yield/force to a bank account is entirely due to real-world side-effects though. Consider a functionally equivalent system such as an e-commerce application with add-stock (deposit) and place-order (withdraw) methods. The invariant is still to make sure the stock of an item never falls below zero. Although concurrent place-order calls may cause the number of items to fall below zero, since after the execution of these methods no tangible goods are given to the clients yet, the yield and force transformations can be applied to these methods.

13.4 Extended Courseware

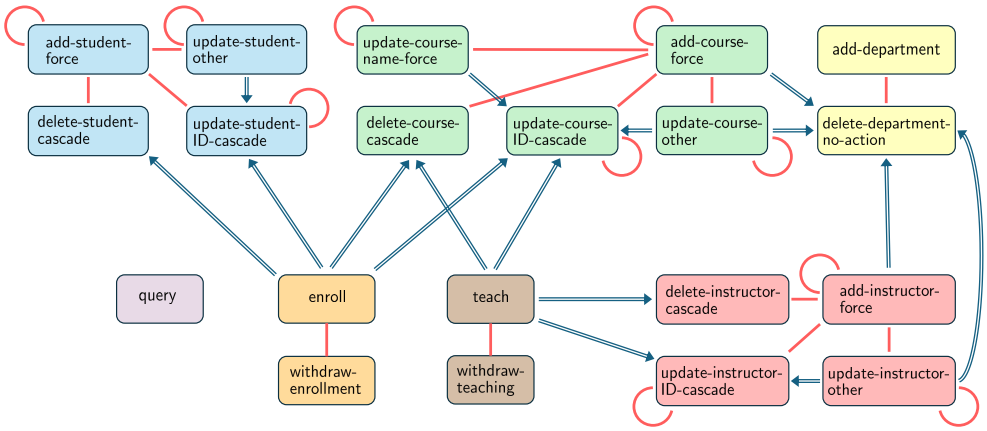


Fig. 14. The conflict graph of the extended courseware relational schema. The abstraction and transformation techniques described in § 7.2 have been applied. Double, directed (blue) edges denote permissibility conflicts. Single, undirected (red) edges denote state conflicts. State-conflict loops are illustrated to present the graph succinctly, but can abstractly be removed by Lemma 3.

We consider a larger, concrete relational schema under the transformations and abstractions presented in § 13.4. Specifically, we extend the courseware schema. We implemented and make this benchmark available, along with its analysis.

The extended courseware benchmark has 6 relations and 20 methods, described below. The conflict graph of this object is shown in Fig. 14. The extended courseware has four independent relations: Student, Instructor, Course, and Department. Further, it has two referential relations: Enrollment (Course, Student) and Teaching (Course, Instructor). In addition to referential integrity constraints, there is a uniqueness constraint on the course name, which is described in more detail below.

- Student. A Student tuple is of the form ⟨ student id, first name, last name ⟩. The student id is the primary key of the Student relation. There are four methods on the Student relation.
  - add-student-force. Inserts/overwrites a student row.
  - delete-student-cascade. Delete the specified student and cascades by removing all enrollment links.
  - update-studentID-cascade. Updates a student id and rewrites enrollment links with the new id.
  - update-student-other. Updates non-primary key attributes of a student, *i.e.*, first and last names.
- Instructor. An Instructor tuple is of the form ⟨ instructor id, first name, last name, department ⟩. The instructor id is the primary key of the Instructor relation. The department is a foreign key to a department in Department. There are four methods on the Instructor relation.
  - add-instructor-force. Inserts/overwrites an instructor row (if the department argument references an existing department).
  - delete-instructor-cascade. Delete the specified instructor and cascades by removing all teaching links.

  - update-instructorID-cascade. Updates an instructor id and rewrites teaching links with the new id.
  - update-instructor-other. Updates non-primary key attributes of an instructor, *i.e.*, first and last names, and department.
- Course. A Course tuple is of the form ⟨ course id, course name, semester, department ⟩. The course id is the primary key of the Course relation. The course name is `UNIQUE`, no two course tuples share the same course name. The department is a foreign key to a department in Department. There are five methods on the Course relation.
  - add-course-force. Inserts/overwrites a course and enforces `UNIQUE` on the course name (if another course already has that name, it is deleted and its enrollment/teaching links are cascaded).
  - delete-course-cascade. Delete the specified course and cascades by removing all enrollment and teaching links.
  - update-courseID-cascade. Updates a course id and rewrites enrollment and teaching links with the new id.
  - update-course-name-force. Updates the specified course name and enforce uniqueness (by deleting any tuple of that course name and cascading its links).
  - update-course-other. Updates the semester or department of a course (if the department argument references an existing department).
- Department. A Department tuple is of the form ⟨ department ⟩. There are two methods on the Department relation.
  - add-department. Inserts a department.
  - delete-department-no-action. Delete the specified department if no tuple references that department.
- Enrollment. The Enrollment relation links students and courses. An Enrollment tuple is of the form ⟨ course id, student id ⟩. The course id and student id are foreign keys to the Course and Student relations, respectively. There are two methods on the Enrollment relation.
  - enroll. Insert a course-student pair (if the course and student arguments references an existing courses and students).
  - withdraw-enrollment. Delete a course-student pair.
- Teaching. The Teaching relation links instructors and courses. A Teaching tuple is of the form ⟨ course id, instructor id ⟩. The course id and instructor id are foreign keys to the Course and Instructor relations, respectively. There are two methods on the Teaching relation.
  - teach. Insert a course-instructor pair (if the course and instructor arguments references an existing courses and instructor).
  - withdraw-teaching. Delete a course-instructor pair.

13.5 NP-Completeness

Theorem 8. co-Tangled *is* NP*-Complete.*

Proof. Firstly, co-Tangled is in NP. Given a certificate π over C, we check that the post-state of every call in π is permissible by simply executing the calls in order of π on σ_0.

Next, we show co-Tangled is NP-Hard with a polynomial-time reduction from the 3COLORING problem. Recall that in 3COLORING, given a graph $G = (V, E)$, the objective is to decide if there exists a coloring $F : V \to \{\text{red}, \text{blue}, \text{green}\}$ such that no two adjacent vertices share a color. Given an instance of 3COLORING, we reduce it to an instance of co-Tangled as follows. The object state is actually the graph G. We denote the state of the object as $H = (V_H, E_H, K_H)$ to disambiguate the two graphs. The map K is a coloring of the vertices. In the initial state σ_0, every vertex is colorless. The invariant $\mathcal{I}$ is that no two adjacent vertices have the same color. There are four methods: red, blue, green, and clear. The red, blue, and green methods take a vertex as input and set it in K_H to the corresponding color. The clear method removes the color of every vertex only if every vertex already has a color. Otherwise, it does nothing. Let the set of calls s be $\{\text{red}(v_i), \text{blue}(v_i), \text{green}(v_i), \text{clear}, \text{clear}\}$, for every v_i in V_H. Clearly, every call in s is permissible in σ_0. Next, we show G has a 3-coloring F iff s is *not* tangled for σ_0.

($\Rightarrow$) Suppose F is a valid 3-coloring for G. Using this assignment, we construct a permutation π of s where every call is permissible in its pre-state. The idea is to color H three times, clearing the colors after the first and second colorings. Let π be composed of three segments. In the first segment, we execute the calls corresponding to their color in F, *e.g.*, if vertex v_i is colored red in F, we execute $\text{red}(v_i)$ in the first segment. Likewise, for blue and green. (The order between these calls is inconsequential.) We include a clear at the end of the first segment.

For the second segment, consider a mapping m from (red, blue, green) to (blue, green, red). (Intuitively, a rotation.) In the second segment, we execute calls on vertices based on their color in F mapped with m, *e.g.*, if vertex v_i is colored red in F, we execute $\text{blue}(v_i)$ in the second segment. We close the second segment with the second and last clear call. The remaining calls compose the third segment; namely, we execute calls on vertices based on their color in F mapped with mapping m' from (red, blue, green) to (green, red, blue).

By routine inspection, we can verify every call in s is included in π exactly once. In the first segment, the vertices of H are colored according to F, before they are cleared. Since F is a valid 3-coloring, then it must be that every call in this segment is permissible. Since the vertex colors are cleared before starting the second segment, by the same reasoning, every call in the second segment is also permissible. This extends to the third segment as well. Thus, every call in π is permissible.

($\Leftarrow$) Assume now that π is a permutation of s such that every call is permissible when executed upon σ_0. From π, we construct a valid 3-coloring F of G. We first claim that at some point in the execution of π, there is some post-state for which every vertex in G has a color. If this claim holds, a valid 3-coloring F immediately follows by simply examining the colors of the vertices in G at this post-state. This is because there are only three available colors, and the integrity property enforces that no two adjacent vertices share a color.

It remains to prove the claim above. Suppose for contradiction there are two vertices v_1 and v_2 in H that are never simultaneously colored. Since every call in s is executed, v_1 and v_2 must each eventually be colored at some point in π. The only way to remove a color from a vertex is using clear. However, clear removes the color of a vertex only if every vertex has a color, implying v_1 and v_2 were simultaneously colored, a contradiction. □

13.6 Experimental Results

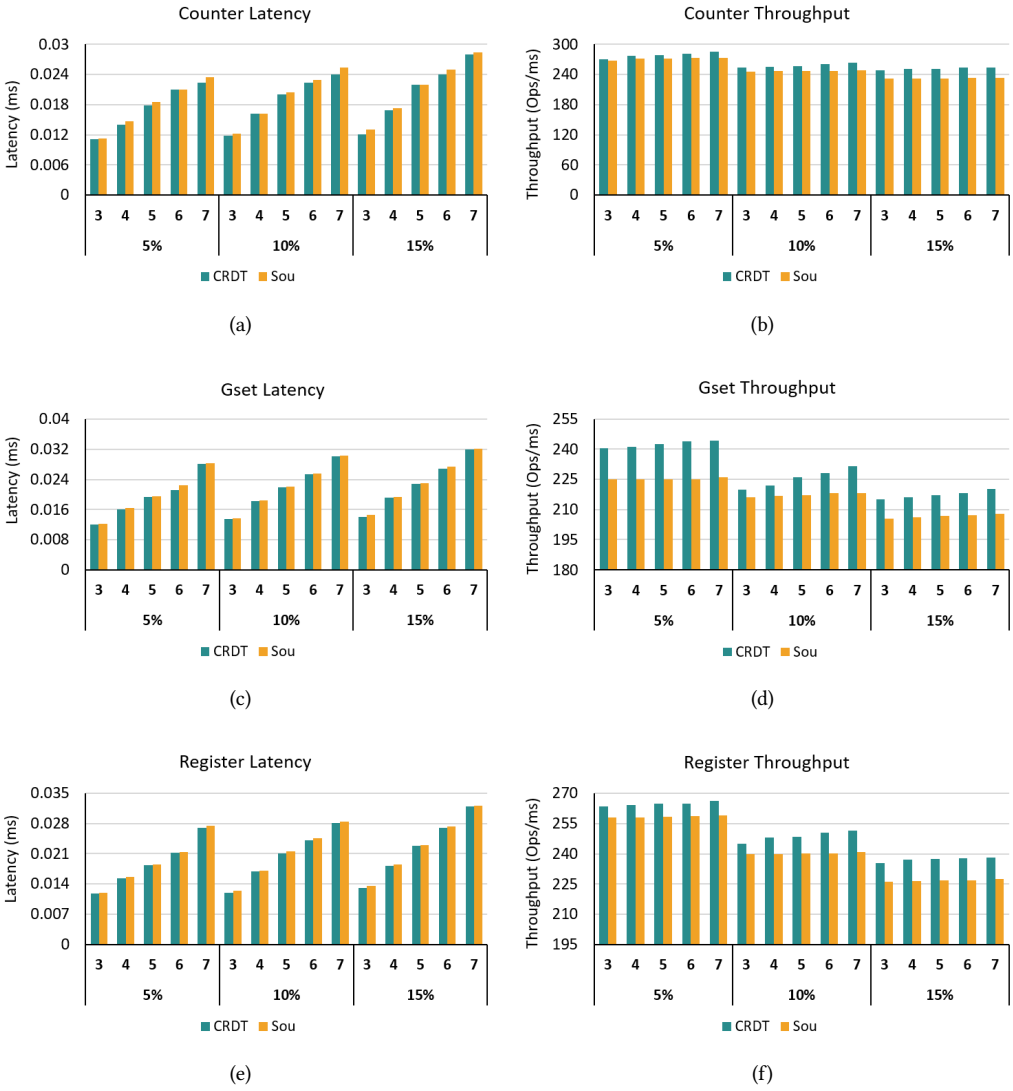


Fig. 15. Optimistic replication of conflict-free use-cases with Frashokereti vs. CRDTs

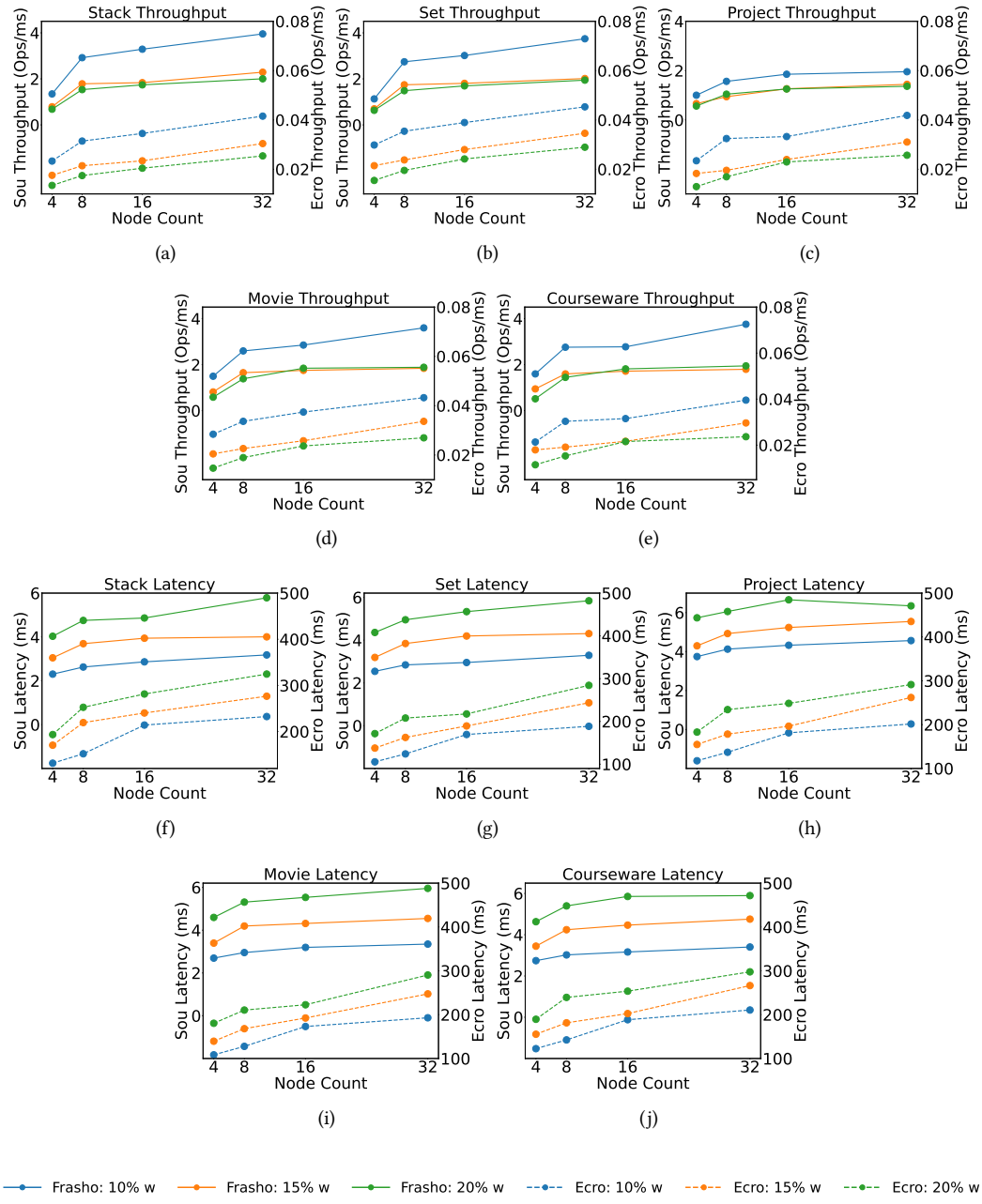
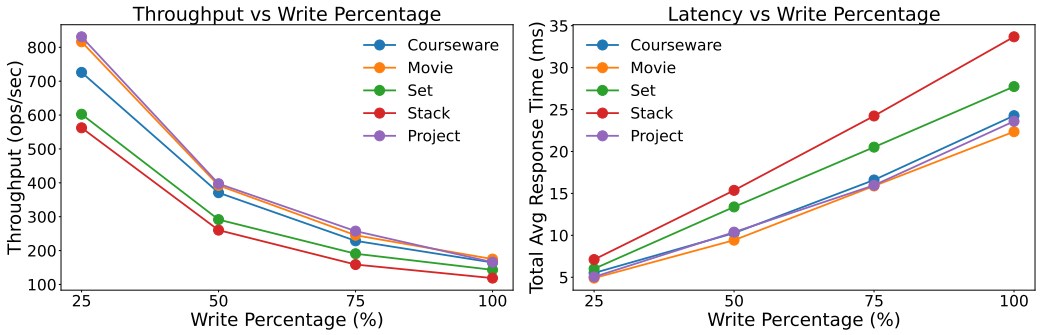
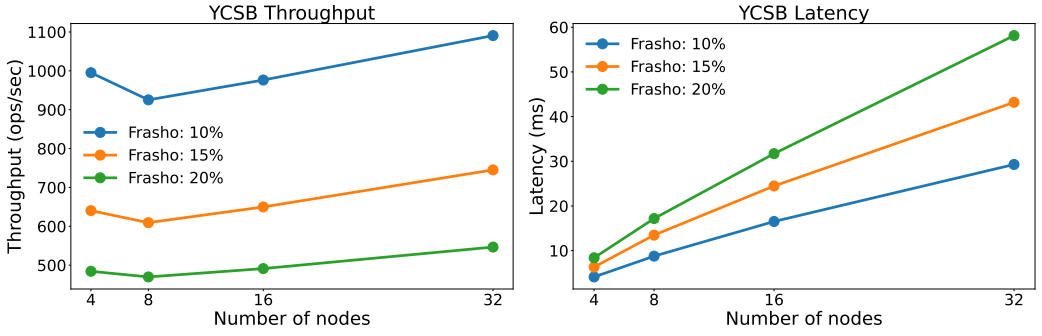


Fig. 16. Optimistic replication of conflicting use-cases with Frashokereti. Note that the left y-axis is for Frashokereti and the right y-axis is for ECRO.



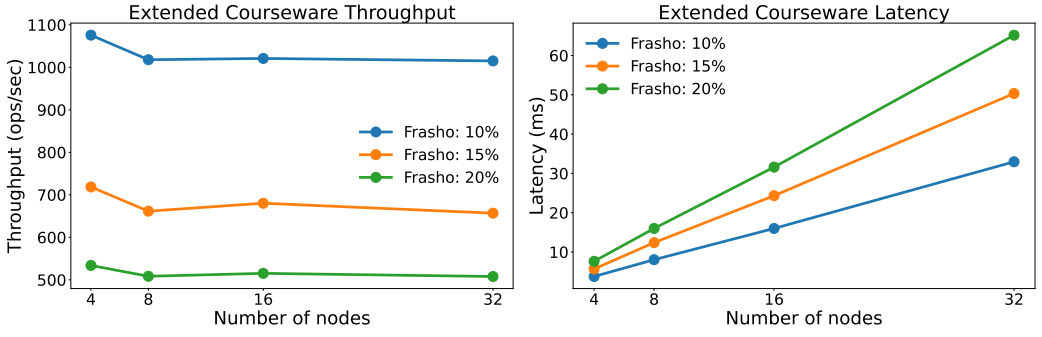
(a) (b)

Fig. 17. write percent



(a) (b)

Fig. 18. ycsb



Fault tolerance experiments

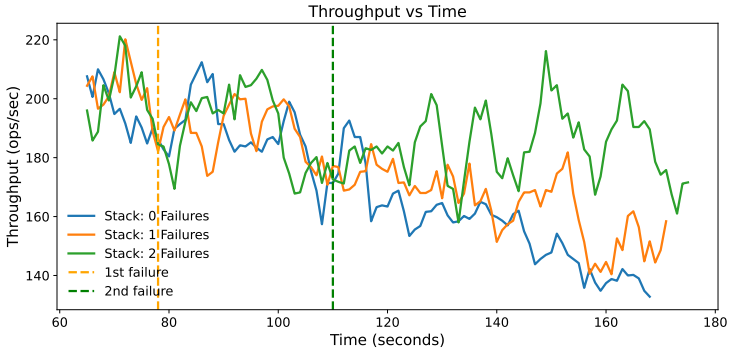


Fig. 19. Throughput vs Time across different failure scenarios.

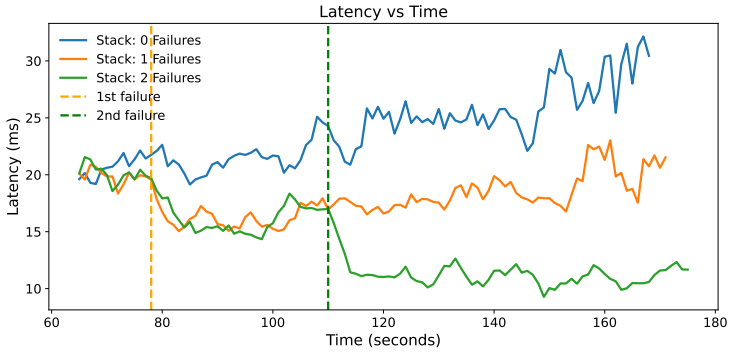


Fig. 20. Latency vs Time across different failure scenarios.

Received 2025-10-10; accepted 2026-02-17

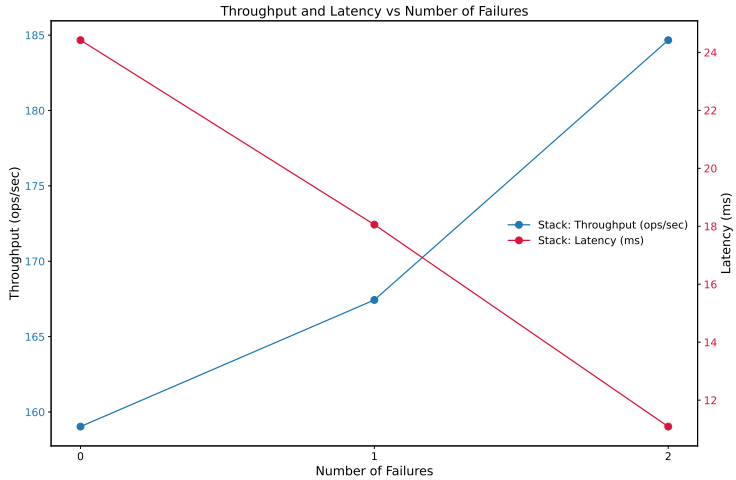


Fig. 21. Average throughput and latency across different failure scenarios, showing the effect of increasing node failures on system performance.