Trillium: Refinement and Higher-Order Distributed Separation Logic

Technical Appendix

ANONYMOUS AUTHOR(S)

A OPERATIONAL SEMANTICS OF ANERISLANG

In AnerisLang, the state σ is of the form $(\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{P}, \mathcal{M})$ where $\mathcal{M}$ is a message soup, a multiset of messages in transit, and $\vec{\mathcal{H}}$, $\vec{\mathcal{S}}$, and $\mathcal{P}$ are mappings from IP addresses to, respectively, node-local heaps $\mathcal{H}$, node-local socket mappings $\mathcal{S}$, and node-local ports in use $\mathcal{P}$. For the relation $(e;\sigma) \to (e';\sigma')$ each step either occurs on the network (a system step) or at some particular node (uniquely determined by its IP address). We define steps that take place on nodes by defining a so-called *head-steps* and closing them under evaluation contexts:

$$\frac{(e,\sigma) \to^h (e',\sigma',(e_{f_1},\cdots,e_{f_k}))}{(K[e];\sigma) \to (K[e'],\sigma',(e_{f_1},\cdots,e_{f_k}))}$$

We split the head step relation into two parts: head steps that do not modify the network-related state (sockets, ports, and the message soup), and the head steps that do modify the network-related state. We call the latter steps for *network-aware* head steps, and in this section we focus on these; the steps that do not modify the network state are mostly standard. Figure 1 shows reduction rules for the network-aware head step $\to_{ip}$ (ip being the IP address of the node executing the head step) — note that none of these steps forks any threads and hence we have omitted forked threads in the form of network-aware head steps. Note how the network-aware head step relation does not have access to the heaps part ($\vec{\mathcal{H}}$) of the state.

A socket mapping $\mathcal{S}$ associates each socket handler (file descriptor) z to a UDP-socket represented by a pair $((ip,p),b)$ and a receiving message buffer $\mathcal{B}$. In the pair $((ip,p),b)$, the address (ip,p) is the one to which the socket handler z is bound, and the Boolean b indicates whether the socket is in a blocking (`true`) or non-blocking (`false`) receive mode. Initially, all sockets are in blocking receive mode (*cf.*, Socket-Bind). The rules Set-Socket-To-Non-Blocking-Receive and Set-Socket-To-Blocking-Receive change the receive mode from blocking to non-blocking using a timeout (n,m) (which represents a float $n.m$).[1] The receive buffer $\mathcal{B}$ stores the set of messages delivered to the node at socket address (ip,p). When a message m is in buffer $\mathcal{B}$ the Receive-Some-Message rule applies and the message m can get delivered to the user application while getting removed from the buffer. When the buffer is empty, one of two network-aware rules applies, depending on whether the socket of the buffer is in the non-blocking (Receive-Empty-Buffer-Non-Blocking) or blocking (Receive-Empty-Buffer-Blocking) receive mode. In the former case, the reduction does not block and it may return `None`. In the latter case, the reduction blocks in the sense that the call to the receive function reduces to itself. Note that his makes the blocking `receivefrom` operation non-atomic. We call such operations stuttering-atomic operations: they reduce to themselves (stutter) a number of times before taking an atomic step. We have shown that the rules applying to atomic steps, *i.e.*, accessing invariants and taking steps in the model, also apply to stuttering atomic steps.

Another way the machine can take a step in AnerisLang is through the system-step relation $\to_{\mathsf{sys}} \subseteq \mathit{State} \times \mathit{State}$. As explained in the Section 2 of the paper, system steps are those that do not correspond to actual program execution steps. The system step relation in AnerisLang has just two

[1] Timeouts with concrete numbers are strictly speaking not needed as we do not model time in AnerisLang; we use timeouts to align `setReceiveTimeout` with the OCaml function Unix.setsockopt_float.

$$\textsc{New-Socket}\quad \frac{z \notin \mathrm{dom}(\mathcal{S}) \qquad \mathcal{S}' = \mathcal{S}[z \mapsto (\mathrm{None}, \emptyset)]}{(\texttt{socket}\ (), (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} (z, (\mathcal{S}', \mathcal{P}, \mathcal{M}))}$$

$$\textsc{Socket-Bind}\quad \frac{\begin{array}{c}\mathcal{S}(z) = (\mathrm{None}, \emptyset) \qquad p \notin \mathcal{P}(ip) \\ \mathcal{S}' = \mathcal{S}[z \mapsto (\mathrm{Some}((ip, p), \texttt{true}), \emptyset)] \\ \mathcal{P}' = \mathcal{P}[ip \mapsto \mathcal{P}(ip) \cup \{p\}]\end{array}}{(\texttt{socketbind}\ z\ (ip, p), (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} (0, (\mathcal{S}', \mathcal{P}', \mathcal{M}))}$$

$$\textsc{Send-Message}\quad \frac{\mathcal{S}(z) = (\mathrm{Some}((ip, p), \texttt{true}), \mathcal{B}) \qquad \mathcal{M}' = \mathcal{M} \uplus \{((ip, p), to, msg)\}}{(\texttt{sendto}\ z\ msg\ to, (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} (|msg|, (\mathcal{S}, \mathcal{P}, \mathcal{M}'))}$$

$$\textsc{Receive-Some-Message}\quad \frac{\begin{array}{c}to = (ip, p) \qquad m = (from, to, msg) \\ \{m\} \in \mathcal{B} \qquad \mathcal{S}(z) = (\mathrm{Some}(to, b), \mathcal{B}) \\ \mathcal{S}' = \mathcal{S}[z \mapsto (\mathrm{Some}(to, b), \mathcal{B} - \{m\})]\end{array}}{(\texttt{receivefrom}\ z, (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} (\texttt{Some}\ (msg, from), (\mathcal{S}', \mathcal{P}, \mathcal{M}))}$$

$$\textsc{Receive-Empty-Buffer-Non-Blocking}\quad \frac{\mathcal{S}(z) = (\mathrm{Some}((ip, p), \texttt{false}), \emptyset)}{(\texttt{receivefrom}\ z, (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} (\texttt{None}, (\mathcal{S}, \mathcal{P}, \mathcal{M}))}$$

$$\textsc{Receive-Empty-Buffer-Blocking}\quad \frac{\mathcal{S}(z) = (\mathrm{Some}((ip, p), \texttt{true}), \emptyset)}{(\texttt{receivefrom}\ z, (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} (\texttt{receivefrom}\ z, (\mathcal{S}, \mathcal{P}, \mathcal{M}))}$$

$$\textsc{Set-Socket-To-Non-Blocking-Receive}\quad \frac{\begin{array}{c}\mathcal{S}(z) = (\mathrm{Some}((ip, p), b), \mathcal{B}) \qquad (0 \leq m \wedge 0 \leq n \wedge 0 < (m + n)) \\ \mathcal{S}'(z) = \mathcal{S}[z \mapsto (\mathrm{Some}((ip, p), \texttt{false}), \mathcal{B})]\end{array}}{(\texttt{setReceiveTimeout}(m, n), (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} ((), (\mathcal{S}', \mathcal{P}, \mathcal{M}))}$$

$$\textsc{Set-Socket-To-Blocking-Receive}\quad \frac{\begin{array}{c}\mathcal{S}(z) = (\mathrm{Some}((ip, p), b), \mathcal{B}) \\ \mathcal{S}'(z) = \mathcal{S}[z \mapsto (\mathrm{Some}((ip, p), \texttt{true}), \mathcal{B})]\end{array}}{(\texttt{setReceiveTimeout}(0, 0), (\mathcal{S}, \mathcal{P}, \mathcal{M})) \rightarrow_{ip} ((), (\mathcal{S}', \mathcal{P}, \mathcal{M}))}$$

Fig. 1. The rules for network-aware head reduction.

steps as seen in Figure 2. The Message-Deliver rule corresponds to the system step that delivers a message $(from, to, msg)$ sent from socket address $from$ to the socket address $to = (ip, p)$. Messages are delivered when they are moved into the corresponding receive buffer. The Message-Drop rule corresponds to the network dropping a message.

Note that the operational semantics of AnerisLang described here is different from the original operational semantics of AnerisLang presented in Krogh-Jespersen et al. [2020]. The difference is that the new operational semantics, as explained above, now features blocking receive operations as

$$\textsc{Message-Deliver}\quad \frac{to = (ip, p) \qquad m = (from, to, msg) \in \mathcal{M} \qquad \vec{\mathcal{S}}(ip) = \mathsf{Some}(\mathcal{S}) \qquad \mathcal{S}(z) = (\mathsf{Some}(to, b), \mathcal{B}) \qquad \vec{\mathcal{S}}' = \vec{\mathcal{S}}[ip \mapsto \mathcal{S}[z \mapsto (\mathsf{Some}(to, b), \mathcal{B} \cup \{m\})]]}{(\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{P}, \mathcal{M}) \to_{\mathsf{sys}} (\vec{\mathcal{H}}, \vec{\mathcal{S}}', \mathcal{P}, \mathcal{M})}$$

$$\textsc{Message-Drop}\quad \frac{m \in \mathcal{M}}{(\vec{\mathcal{H}}, \vec{\mathcal{S}}, \mathcal{P}, \mathcal{M}) \to_{\mathsf{sys}} (\vec{\mathcal{H}}, \vec{\mathcal{S}}', \mathcal{P}, \mathcal{M} \setminus \{m\})}$$

Fig. 2. The rules for configuration head reduction.

well as separate system steps for dropping and delivering messages. The old operational semantics did not remove dropped messages from the message soup; it simply ignored them. Also, the old operational semantics did not feature buffers for sockets; any message in the message soup could simply be delivered upon performing a receive operation. The motivation for these changes is to make the operational semantics more realistic.

B SEMANTICS OF WEAKEST PRECONDITIONS IN TRILLIUM

The core idea of Trillium is to track a model trace alongside the program execution trace and *enforce* that whenever the program takes a step, there is an extension of the model trace that corresponds to this step. This is achieved using a new weakest precondition, from which one can easily recover Hoare triples.

The weakest precondition of Trillium is defined using the Iris base logic, similarly to how the weakest precondition of the Iris program logic is defined using the Iris base logic [Jung et al. 2018]. We recall that the Iris base logic is a higher-order logic equipped with facilities for definitions and reasoning by guarded recursion and with support for reasoning about ownership (separation logic) and invariants. Concretely, the Trillium definition of the weakest precondition predicate $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$ is as follows:

$$\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\} \triangleq \begin{cases} \Rrightarrow_{\mathcal{E}} \Phi(e) & \text{if } e \in \mathit{Val} \\ \forall \tau, \mu, i, K. \\ \quad \mathit{thread}(i, \mathit{last}(\tau)) = K[e] \mathrel{-\!\!*} \\ \quad \mathit{ValidExec}(\tau) \mathrel{-\!\!*} \\ \quad \mathsf{StateInterp}(\tau, \mu)\, {}^{\mathcal{E}}\!\Rrightarrow^{\emptyset} \\ \quad \mathit{reducible}(e, \mathit{state}(\mathit{last}(\tau))) * \\ \quad \forall e', \sigma', (e_{f_1}, \cdots, e_{f_k}). & \text{otherwise} \\ \quad\quad (e, \mathit{state}(\mathit{last}(\tau))) \to (e', \sigma', (e_{f_1}, \cdots, e_{f_k})) \mathrel{-\!\!*} \triangleright\, {}^{\emptyset}\!\Rrightarrow^{\mathcal{E}} \\ \quad\quad \exists \delta. \mathsf{StateInterp}(\tau ::_{tr} \mathit{update}_{th}(i, K[e'], \sigma', \mathit{last}(\tau)), \mu ::_{tr} \delta) * \\ \quad\quad\quad \mathsf{ValidEvolution}(\tau, \mu, \mathit{update}_{th}(i, K[e'], \sigma', \mathit{last}(\tau)), \delta) * \\ \quad\quad\quad \mathsf{wp}_{\mathcal{E}}\, e'\, \{\Phi\} * \mathop{\Large *}_{1 \le j \le k} \mathsf{wp}_{\top}\, e_{f_j}\, \{\mathsf{True}\} \end{cases}$$

Here the postcondition Φ is a predicate that takes a return value as an argument. We sometimes write Φ as $x.\, P$; then x acts as a binder for the return value in P. In the above definition, Iris-specific logical connectives are typeset in blue; to understand the high-level ideas of the definition and, in

particular, what is new compared to the Iris definition of weakest preconditions, they can mostly be ignored. On a first reading $\Rightarrow_{\mathcal{E}}$ and $\triangleright$ may be ignored, and $*$ may be thought of as ordinary conjunction, and both $-\!\!*$ and ${}^{\mathcal{E}}\!\Rightarrow\!\!*^{\mathcal{E}'}$ as ordinary implication.

Next we remark that, just as for standard Iris weakest preconditions, our definition implies safety: if $\mathsf{wp}\ e\ \{\Phi\}$ holds then e will not get stuck and whenever it reduces to a value, then that value satisfies the postcondition. This follows from the high-level pattern of the definition: either e is a value in which case the postcondition must hold, or e is reducible in the current state (the state of the last configuration in our execution trace τ) and, furthermore, whatever e reduces to in this current state (as well as all the forked threads) should again satisfy the weakest precondition—we do not care about the postcondition of the forked threads.

The parts typeset in red and green in the definition above are, respectively, the parts that have been added or adapted, compared to standard Iris weakest preconditions. We now explain these parts in more detail.

The state interpretation predicate, StateInterp, in Iris weakest preconditions takes only the current state as an argument; here it has been extended to take the entire execution trace of the program as well as the model trace as arguments. This is crucial and means in particular that we can reflect the history of the program execution in the program logic and use it in our reasoning. For example, in the AnerisLang instantiation of Trillium, it allows us to track the history of sent and received messages. In the paper we discuss how this is useful for expressing the correctness of CRDTs. The role of the StateInterp predicate is to tie the state of both the program and the model state to Iris resources, *e.g.*, the points-to predicate of separation logic for heap locations; see Trillium's instantiation with AnerisLang in the main paper for more details. To appreciate the role of the StateInterp predicate better, let us consider part of the soundness proof of the following Aneris inference rule:[2]

$$\textsc{wp-load}\quad \frac{\ell \mapsto_{ip} v}{\mathsf{wp}_{\mathcal{E}}\ \langle ip;\ !\,\ell\rangle\ \{x.\ x = v * \ell \mapsto_{ip} v\}}$$

This rule states that if we own the location ℓ with value v on a node with IP address ip, *i.e.* we have the points-to predicate $\ell \mapsto_{ip} v$, then on that node, reading ℓ returns a value that is equal to v and, moreover, we retain ownership of ℓ. To show soundness of this rule, we need to show that the program $!\,\ell$ does not get stuck when run in the *current state* which would happen if ℓ was not allocated. Hence, we need to rule out that case and show that no matter what the current state may be, ℓ is allocated in that state—note how the current state (in our case part of τ) is universally quantified in the definition of the weakest precondition predicate. This is where StateInterp plays an important role: StateInterp and the points-to predicate are defined so that they satisfy the following property:[3]

$$\mathsf{StateInterp}(\tau, \mu) * \ell \mapsto_{ip} v \vdash \mathit{Heap}(ip, \mathit{last}(\tau))(\ell) = v$$

where $\mathit{Heap}(ip, c)$ is the heap (a partial map with finite support from locations to values) for the node with IP address ip in the state of the configuration c. Here $\vdash$ is Iris's entailment relation. Hence, we can conclude that ℓ is allocated in the heap of the current state and that the program does not get stuck.

Apart from knowing that the resources pertaining to state, such as heap points-to predicates, network messages, *etc.*, are properly reflected in Iris resources, we need to know that the execution

[2]In the AnerisLang instantion of Trillium, a program expression consists of a pair of a node IP address and an ordinary expression.

[3]See Jung et al. [2018] for details of how this can be done in Iris.

trace τ is consistent with a given program. That is, when proving wp $e\,\{\Phi\}$ we need to know that whatever the execution trace τ up to now is, the expression e is now about to take a step. This is captured by $\mathit{thread}(i, \mathit{last}(\tau)) = K[e]$, which states that on thread i of the current configuration, $\mathit{last}(\tau)$, the expression e can now take a step, *i.e.*, e appears under an evaluation context K.

The definition of the weakest precondition requires that after e takes a step, there must be a model state δ such that the following holds:

(1) The state interpretation holds for the resulting execution trace and model trace (note that both traces are extended with their respective new states).
(2) The user specified predicate ValidEvolution is satisfied.
(3) The weakest precondition holds for e', the program that e steps to.
(4) All threads forked during the step also satisfy the weakest precondition.

Note also that the user of Trillium can restrict the new model state δ by picking ValidEvolution appropriately.

In summary, the weakest precondition wp $e\,\{\Phi\}$ states that e is safe to execute (does not get stuck), that any value resulting from e satisfies the postcondition Φ, and, that every execution step has been matched by a model step.

C TWO-PHASE COMMIT

The two-phase commit protocol [Gray 1978] is one of the best-known practical algorithms for solving the *transaction-commit problem* where a collection of processes, called *resource managers*, have to agree on whether a transaction ought to be *committed* or *aborted*. A protocol that solves the problem has to ensure *agreement*, *i.e.*, no two processes may decide differently.

In this development, we consider a TLA$^+$ model of transaction commit and show that a distributed implementation of the two-phase commit protocol refines it. As a corollary of the refinement and the agreement theorem for the model we show that the implementation also ensures agreement.

Model. The transaction commit model is summarized in Figure 3. The model is parameterized by a set *RMs* of resource managers that are each in either an initial Working state, a preparation state Prepared, or in a final state Committed or Aborted. The full state of the model is a finite mapping from resource managers to one of these states. The transition relation allows resource managers to transition freely from the Working to the Prepared state (TC-Prepare). A resource manager may transition from the Prepared to the Committed state if all resource managers are either in the Prepared or the Committed state (TC-Commit), and from the Working or Prepared state to the Aborted state if no resource manager is in the Committed state (TC-Abort).

By induction on the transition relation one can easily show that the model satisfies the agreement property when starting from an initial state where all resource managers are in the Working state.

Theorem C.1 (Agreement of TC). *Let* $\delta_{\mathsf{Init}}(r) \triangleq \mathsf{Working}$. *If* $\delta_{\mathsf{Init}} \rightarrow^*_{\mathsf{TC}} \delta$ *then for all* $r_1, r_2 \in \mathit{RMs}$ *it is* not *the case that* $\delta(r_1) = \mathsf{Committed}$ *and* $\delta(r_2) = \mathsf{Aborted}$.

Implementation. The two-phase commit protocol relies on a *transaction manager* to orchestrate the agreement process; the transaction manager may either be a distinguished resource manager or a separate process. Listing 1 and Listing 2 show implementations in AnerisLang of the transaction manager and resource manager roles, respectively. The transaction manager uses a simple library functionality for removing duplicate messages; the functionality is initialized with the `nodup_init` invocation that returns a wrapped `receivefrom` primitive that removes duplicates messages but is otherwise not important.

The transaction manager implementation starts by allocating a socket and binding it to the socket address `TM` given as argument. It continues by sending a `"PREPARE"` message to all the resource

$$\text{RMStates} \triangleq \{\text{Working}, \text{Prepared}, \text{Committed}, \text{Aborted}\}$$

$$\delta \in RMs \xrightarrow{\text{fin}} \text{RMStates}$$

$$\text{CanCommit}(\delta) \triangleq \forall r \in RMs.\, \delta(r) = \text{Prepared} \vee \delta(r) = \text{Committed}$$

$$\text{NotCommitted}(\delta) \triangleq \forall r \in RMs.\, \delta(r) \neq \text{Committed}$$

$$\text{TC-Prepare}\quad \frac{\delta(r) = \text{Working}}{\delta \rightharpoonup_{\text{TC}} \delta[r \mapsto \text{Prepared}]} \qquad \text{TC-Commit}\quad \frac{\delta(r) = \text{Prepared} \qquad \text{CanCommit}(\delta)}{\delta \rightharpoonup_{\text{TC}} \delta[r \mapsto \text{Committed}]}$$

$$\text{TC-Abort}\quad \frac{\delta(r) = \text{Working} \vee \delta(r) = \text{Prepared} \qquad \text{NotCommitted}(\delta)}{\delta \rightharpoonup_{\text{TC}} \delta[r \mapsto \text{Aborted}]}$$

Fig. 3. TLA$^+$ specification of the transaction-commit (TC) problem.

Listing 1. Transaction manager.

```
let recv_resps recv skt RMs =
  let rec loop prepared =
    Set.equal prepared RMs ||
      let (msg, sndr) = recv skt in
      msg = "PREPARED" &&
        loop (Set.add sndr prepared) in
  loop (Set.empty ()) in

let transaction_manager TM RMs =
  let skt = socket () in
  socketbind skt TM;
  let recv = nodup_init () in
  sendto_all skt RMs "PREPARE";
  let ready = recv_resps recv skt RMs in
  if ready then
    sendto_all skt RMs "COMMIT";
    receivefrom_all skt recv RMs;
    "COMMITTED"
  else
    sendto_all skt RMs "ABORT";
    "ABORTED"
```

Listing 2. Resource manager.

```
let resource_manager RM TM =
  let skt = socket () in
  socketbind skt RM;
  let (m, _) = receivefrom skt in
  if m = "ABORT"
  then sendto skt "ABORTED" TM
  else
    let local_abort = coin_flip () in
    if local_abort
    then sendto skt "ABORTED" TM
    else
      sendto skt "PREPARED" TM;
      let (decision, _) =
        wait_receivefrom skt
        (fun (_, m) => m = "COMMIT" ||
                       m = "ABORT") in
      if decision = "COMMIT" then
        sendto skt "COMMITTED" TM
      else
        sendto skt "ABORTED" TM
```

manager socket addresses given in `RMs`, asking the resource managers to transition to the preparation phase. If all the resource managers respond with `"PREPARED"`—signifying that they are all ready to commit—the transaction manager continues by sending a `"COMMIT"` message, telling the resource managers the decision is to commit, after which it awaits their responses and returns. If a single resource manager responds with `"ABORTED"`, the transaction manager stops receiving responses, relays the information, and returns.

The resource manager implementation starts by allocating a socket and binding it to the socket address `RM` given as argument. It continues by listening for an initial request from the transaction manager; in case another resource manager already aborted and this information arrived prior to the initial `"PREPARE"` request, the resource manager aborts. If asked to prepare, the resource manager makes a local decision—here with a nondeterministic coin flip–and sends the decision to

the transaction manager. If the resource manager decides to abort, it immediately returns; otherwise it awaits the final decision from the transaction manager, confirms the transition, and returns.

Refinement. To show that the two-phase commit implementation refines the transaction-commit model we instantiate the Aneris logic with the model; this gives us a handle to the current model state δ that we can manipulate through the separation logic resource $\mathsf{Model}_\circ(\delta)$. The key proof strategy is to keep this resource in an invariant that ties together the model state and the physical with enough information such that the continued simulation is strong enough for proving our final correctness theorem (Theorem C.2). In this development, we will tie sending a message (such as `"COMMITTED"`) from resource manager r to the corresponding transition in the model (such as TC-Commit). Additionally, the invariant will have to keep sufficient ghost resources and information for us to establish the conditions ($\mathrm{CanCommit}(\delta)$ and $\mathrm{NotCommitted}(\delta)$) for progressing the model.

To state a sufficient invariant for the two-phase commit refinement we will make use of two resource algebras: a variation of the *oneshot* algebra [Jung et al. 2018] with *discardable fractions* [Vindum and Birkedal 2021] as well as a monotone ghost map algebra.

The oneshot algebra with discardable fractions allows us to define resources $\mathit{pending}(q)$, $\mathit{discarded}$, and $\mathit{shot}(a)$ governed by the rules below.

$$\mathit{pending}(q) \Rrightarrow\!\!* \mathit{discarded} \qquad \mathit{pending}(1) \Rrightarrow\!\!* \mathit{shot}(a)$$
$$\mathit{shot}(a) * \mathit{pending}(q) \vdash \mathsf{False} \qquad \mathit{pending}(p) * \mathit{pending}(q) \dashv\vdash \mathit{pending}(p+q)$$
$$\mathit{shot}(a) * \mathit{discarded} \vdash \mathsf{False} \qquad \mathit{shot}(a) * \mathit{shot}(a) \dashv\vdash \mathit{shot}(a)$$
$$\mathit{shot}(a) * \mathit{shot}(b) \vdash a = b \qquad \mathit{discarded} * \mathit{discarded} \dashv\vdash \mathit{discarded}$$

Intuitively, $\mathit{pending}(q)$ corresponds to owning a q-sized share in making some decision; only by owning all shares a unique decision can be made as witnessed by owning $\mathit{shot}(a)$. By discarding a share a party can ensure that no decision can ever be made. Notice how this construction can be used to model the two-phase commit protocol (in particular the condition $\mathrm{CanCommit}(\delta)$ and $\mathrm{NotCommitted}(\delta)$) by picking the decision value a to be the unit value: each party initially owns an evenly sized share of the decision and transfers this share to the transaction manager when preparing to commit. By receiving a share from all resource managers, the transaction manager can make the decision to commit. By discarding a share, a resource manager can ensure that no decision to commit will ever be made and safely abort.

Using the *monotone resource algebra* [Timany and Birkedal 2021], we construct a logical points-to connective $r \overset{q}{\mapsto}_\bullet s$ that will track q-fractional ownership of the current model state s of resource manager r, but where s may only evolve monotonically according to the internal resource manager transition relation given by Working $\rightsquigarrow$ Prepared $\rightsquigarrow$ Committed and Working, Prepared $\rightsquigarrow$ Aborted. The construction is accompanied by a duplicable $r \mapsto_\circ s$ resource that gives a *lower-bound* on the current state of resource manager r as seen from the rules below.

$$r \overset{q}{\mapsto}_\bullet s * r \mapsto_\circ s' \vdash s' \rightsquigarrow^* s$$
$$r \overset{1}{\mapsto}_\bullet s * s \rightsquigarrow^* s' \Rrightarrow\!\!* r \overset{1}{\mapsto}_\bullet s' * r \mapsto_\circ s'$$
$$r \mapsto_\circ s * r \mapsto_\circ s \dashv\vdash r \mapsto_\circ s$$

Equipped with the two constructions from above we can define the refinement invariant for the two-phase commit implementation:

$$I_{\mathsf{TPC}} \triangleq \exists \delta.\, \mathsf{Model}_\circ(\delta) * \mathop{\Large *}_{r \in RMs} \exists R, T, s.\; \begin{array}{l} r \overset{\frac{1}{2}}{\mapsto}_\bullet s * \delta(r) = s * \mathrm{TokenCoh}(s) * \\ r \rightsquigarrow^{\phi_{\mathsf{RM}}}_\square (R, T) * \mathrm{ModelCoh}(r, s, T) \end{array}$$

The invariant owns the current model state δ and for each resource manager r it owns half of the corresponding monotone points-to connective for some state s such that $\delta(r) = s$; the resource manager itself will own the remaining half. This ensures that the resource manager itself knows exactly which state it is in and that the resource cannot be updated without updating the model as well. We moreover tie being in the model states Committed and Aborted to ownership of, respectively, the *shot* and *discarded* resources as given by $\mathrm{TokenCoh}(s)$ below.

$$\mathrm{TokenCoh}(s) \triangleq \begin{cases} \mathit{shot} & \text{if } s = \mathsf{Committed} \\ \mathit{discarded} & \text{if } s = \mathsf{Aborted} \\ \mathsf{True} & \text{otherwise} \end{cases}$$

The remaining two clauses constitute the key component in connecting the model to the physical state; the persistent socket protocol $r \rightsquigarrow^{\phi_{\mathsf{RM}}}_\square (R, T)$ tracks the history T of sent messages from resource manager r and $\mathrm{ModelCoh}(r, s, T)$ requires that if the resource manager r is in state s then a corresponding message must have been sent to the transaction manager t and if a message corresponding to a state s' has been sent, the resource manager must be in *at least* that state:

$$\mathrm{MessageCoh}(r, s, T) \triangleq \begin{cases} (r, t, \text{"PREPARED"}) \in T & \text{if } s = \mathsf{Prepared} \\ (r, t, \text{"COMMITTED"}) \in T & \text{if } s = \mathsf{Committed} \\ (r, t, \text{"ABORTED"}) \in T & \text{if } s = \mathsf{Aborted} \\ \mathsf{True} & \text{otherwise} \end{cases}$$

$$\mathrm{ModelCoh}(r, s, T) \triangleq \mathrm{MessageCoh}(r, s, T) \land \forall s'.\, \mathrm{MessageCoh}(r, s', T) \longrightarrow s' \rightsquigarrow^* s$$

The socket protocol ϕ_{TM} governing the communication with the transaction manager is defined below. It follows the intuitive description given earlier: when preparing to commit, the *pending* resource is transferred to the transaction manager, and in order to commit or abort, the resources *shot* and *discarded* must be transferred as well, respectively. Moreover, the resource manager has to prove that its model state has (at least) been progressed to the corresponding states. The socket protocol for ϕ_{RM} for the resource manager follows a similar pattern.

$$\begin{aligned} \phi_{\mathsf{TM}}(r, t, b) \triangleq\; & r \in RMs * \\ & \Big(b = \text{"PREPARED"} * \mathit{pending}(\tfrac{1}{|RMs|+1}) * r \mapsto_\circ \mathsf{Prepared}\Big) \lor \\ & (b = \text{"COMMITTED"} * \mathit{shot} * r \mapsto_\circ \mathsf{Committed}) \lor \\ & (b = \text{"ABORTED"} * \mathit{discarded} * r \mapsto_\circ \mathsf{Aborted}) \\ \phi_{\mathsf{RM}}(r, t, b) \triangleq\; & b = \text{"PREPARE"} \lor \\ & (b = \text{"COMMIT"} * \mathit{shot} * \mathop{\Large *}_{r \in RMs} r \mapsto_\circ \mathsf{Prepared}) \lor \\ & (b = \text{"ABORT"} * \mathit{discarded}) \end{aligned}$$

The transaction manager implementation can be given the specification below; notice how it does not rely on the refinement invariant but only on the socket protocols and resources as described.

$$\left\{\begin{array}{l}\mathsf{Fixed}(A) * t \in A * \mathsf{FreePort}(t) * t \rightsquigarrow (\emptyset, \emptyset) * \\ \mathit{pending}(\frac{1}{|RMs|+1}) * t \Rrightarrow \phi_{\mathsf{TM}} * \text{✻}_{r \in RMs}\, r \Rrightarrow \phi_{\mathsf{RM}}\end{array}\right\}$$
$$\langle t;\ \texttt{transaction\_manager}\ t\ RMs\rangle$$
$$\left\{v.\ \begin{array}{l}(v = \text{"COMMITTED"} * \text{✻}_{r \in RMs}\, r \mapsto_\circ \mathsf{Committed}) \vee \\ (v = \text{"ABORTED"} * \exists r \in RMs.\, r \mapsto_\circ \mathsf{Aborted})\end{array}\right\}$$

The specification for the resource manager as seen below, however, relies on the invariant as well as fractional ownership of the resource manager's model state.

$$\left\{\begin{array}{l}\mathsf{Fixed}(A) * r \in A * \mathsf{FreePort}(r) * \boxed{I_{\mathsf{TPC}}}^{\mathcal{N}_{\mathsf{TPC}}} * \\ \\ r \Rrightarrow \phi_{\mathsf{RM}} * t \Rrightarrow \phi_{\mathsf{TM}} * \mathit{pending}(\frac{1}{|RMs|+1}) * r \overset{\frac{1}{2}}{\mapsto}_\bullet \mathsf{Working}\end{array}\right\}$$
$$\langle r;\ \texttt{resource\_manager}\ r\ t\rangle$$
$$\{\mathsf{True}\}$$

Theorem C.2 (Agreement, two-phase commit implementation). *If* $(e;\emptyset) \to^* (T;\sigma)$ *and* $m_{s_1}, m_{s_2} \in \mathcal{M}$ *such that* m_{s_i} *is the physical message corresponding to state* s_i *then it is not the case that* $s_1 = \mathsf{Committed}$ *and* $s_2 = \mathsf{Aborted}$.

D SINGLE-DECREE PAXOS

D.1 Auxiliary implementation components

```
let recv_promises skt n bal0 =
  let promises = ref (Set.empty ()) in
  let senders = ref (Set.empty ()) in
  let rec loop () =
    if Set.cardinal !senders = n
    then !promises
    else
      let (m, sndr) = receivefrom skt in
      let (bal, mval) =
        proposer_deser m in
      if bal = bal0 then
        senders <- Set.add !senders sndr;
        promises <- Set.add !promises mval
      else ();
      loop ()
  in loop ()
```

```
let find_max_promise s =
  let max_promise acc promise =
    match promise, acc with
    | Some (b1, _), Some (b2, _) =>
      if b1 < b2 then acc else promise
    | None, Some _ => acc
    | _, _ => promise
    end
  in Set.fold max_promise s None
```

```
let learner acceptors addr client =
  let skt = socket () in
  socketbind skt addr;
  let majority =
    Set.cardinal acceptors / 2 + 1 in
  let votes = ref (Map.empty ()) in
  let rec go () =
    let (m, sndr) = receivefrom skt in
    let (bal, v) = learner_deser m in
    let bal_votes =
      match Map.find_opt bal !votes with
      | Some vs => vs
      | None => Set.empty ()
      end in
      let bal_votes' =
        Set.add sndr bal_votes in
    if Set.cardinal bal_votes' = majority
    then (bal, v)
    else
      votes <- Map.add bal bal_votes' votes;
      go () in
    let result = go () in
    let _ = sendto skt
      (client_ser result) client in
    result
```

```
let wait_receivefrom skt test =
  let rec loop () =
    let msg = receivefrom skt in
    if test msg then msg else loop ()
  in loop ()

let sendto_all skt X msg =
  Set.iter (fun x => sendto skt msg x) X
```

E FAIR TERMINATION OF CONCURRENT PROGRAMS, DETAILS

The section gives a few definitions which were alluded to in Section 5.

E.1 Fairness model

A fairness model is a state transition system, with a set of roles which label its transitions. Each state has a set of enabled roles, and a "fuel limit" which is used to keep the control the branching of Live($\mathcal{F}$) which we define below.

Definition E.1. A *fairness model* $\mathcal{F}$ is the data of a set $\mathcal{F}$ of states, a set $\mathcal{R}$ of roles, and a transition relation $\rightarrow \subseteq \mathcal{F} \times \mathcal{R} \times \mathcal{F}$ labeled by roles. Moreover, it is equipped with a map *enabled_roles* : $\mathcal{F} \rightarrow \wp_{\text{fin}}(\mathcal{R})$ which associates a finite set of *enabled roles* to each state $s \in \mathcal{F}$. It must approximate the set of outgoing roles:

$$\forall \rho \in \mathcal{R},\ \forall \delta, \delta' \in \mathcal{F},\ \delta \xrightarrow{\rho} \delta' \implies \rho \in \mathit{enabled\_roles}\ \delta.$$

and it must not disable other roles: if $\delta \xrightarrow{\rho} \delta'$, then

$$\forall \rho' \neq \rho, \quad \rho' \in \mathit{enabled\_roles}\ \delta \implies \rho' \in \mathit{enabled\_roles}\ \delta'$$

Finally, $\mathcal{F}$ comprises a map *fuel_limit* : $\mathcal{F} \rightarrow \mathbb{N}$ which will be useful to ensure finite branching conditions.

A *run*, or *trace* of $\mathcal{F}$ is a non-empty finite or infinite sequence of the form:

$$\delta_1 \xrightarrow{\rho_1} \delta_2 \xrightarrow{\rho_2} \cdots$$

E.2 The Live construction

Given a "fairness model" $\mathcal{F}$, we define a (labeled) STS Live($\mathcal{F}$) which keeps track of mapping between roles and threads, and of fuels, as explained in Section 5.

A state of $\mathsf{Live}(\mathcal{F})$ is a triple (δ, F, T) of a state $\delta \in \mathcal{F}$, together with two maps $F : \mathit{enabled\_roles}\ \delta \to \mathbb{N}$ and $T : \mathit{enabled\_roles}\ \delta \to \mathbb{N}$ which associate a fuel amount and a thread id to each role ρ which is enabled in the current underlying state δ.

The set of labels of $\mathsf{Live}(\mathcal{F})$ is

$$\{\mathsf{Step}\ \rho\ \mathsf{tid} \mid \rho \in \mathcal{R}, \mathsf{tid} \in \mathbb{N}\} \quad \uplus \quad \{\mathsf{Silent}\ \mathsf{tid} \mid \mathsf{tid} \in \mathbb{N}\}$$

(recall that $\mathcal{R}$ is the set of roles of the "fairness model" $\mathcal{F}$) The intuition is that a step labeled by $\mathsf{Step}\ \rho\ \mathsf{tid}$ corresponds to the situation where the thread tid takes a step in the program, and one of the roles ρ under its responsibility takes a step in the fairness model. A step labeled $\mathsf{Silent}\ \mathsf{tid}$, on the other hand, corresponds to a step in the program which does not correspond to a step in the fairness model, in other words, a stuttering step.

We now describe the transitions in the labeled STS $\mathsf{Live}(\mathcal{F})$. The idea is that there are two kinds of transitions

- A thread tid can take a step in $\mathsf{Live}(M)$ of the form
$$(\delta, F, T) \xrightarrow{\mathsf{Silent\ tid}} (\delta, F', T')$$
which does not corresponds to a step in the underlying fairness model $\mathcal{F}$ in exchange of consuming fuel: for *every* $\rho \in T^{-1}(\mathsf{tid})$ which the thread tid is in charge of, the corresponding fuel $F(\rho)$ must decrease strictly, in that $F'(\rho) < F(\rho)$.
- A thread tid can also take a step in the underlying model which corresponds to a role $\rho \in T^{-1}(\mathsf{tid})$
$$(\delta, F, T) \xrightarrow{\mathsf{Step}\ \rho\ \mathsf{tid}} (\delta', F', T')$$
This allows to refill the fuel of ρ up to the limit $\mathit{fuel\_limit}\ \delta'$, that is, $F'(\rho) \leq \mathit{fuel\_limit}\ \delta'$; this is required to keep the STS finitely branching. All the other roles which are associated with tid must decrease:
$$\forall \rho' \in T(\mathsf{tid}) \setminus \{\rho\}, \quad F'(\rho') < F(\rho')$$
Roles which appear between δ and δ' can have any fuel $\leq \mathit{fuel\_limit}\ \delta'$ in F'. Of course, we also require there be a step
$$\delta \xrightarrow{\rho} \delta'$$
in the fairness model $\mathcal{F}$.

In addition to the two constraints above, in both cases, the fuel of the roles which are not associated with the thread tid must not increase:

$$\forall \rho \in \mathcal{R} \setminus T^{-1}(\mathsf{tid}), \quad F'(\rho) \leq F(\rho)$$

and roles which change owners ($T'(\rho) \neq T(\rho)$) must have their fuel decrease strictly.

E.3 The $\mathcal{F}_{\mathrm{YN}}$ model

The full $\mathcal{F}_{\mathrm{YN}}$ model is defined as follows. Its states are quadruples

$$(m, b, ye, ne) \in \mathbb{N} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}$$

Its set of roles is $\{\mathsf{Yes}, \mathsf{No}\}$, *fuel_limit* is the constant map equal to 30. The map *enabled_roles* is defined so that

$$\mathsf{Yes} \in \mathit{enabled\_roles}\ (m, b, ye, ne) \iff ye = 1$$
$$\mathsf{No} \in \mathit{enabled\_roles}\ (m, b, ye, ne) \iff ne = 1$$

It remains to define the transitions. First, there are the main types of transition which is described in Figure 7:

(1) success for Yes: $(m, 1, 1, 1) \xrightarrow{\text{Yes}} (m, 0, 1, 1)$ if $m > 0$
(2) failure for Yes: $(m, 0, 1, 1) \xrightarrow{\text{Yes}} (m, 0, 1, 1)$ if $m > 0$
(3) success for No: $(m, 0, 1, 1) \xrightarrow{\text{No}} (m - 1, 1, 1, 1)$ if $m > 0$
(4) failure for No: $(m, 1, 1, 1) \xrightarrow{\text{No}} (m, 1, 1, 1)$

and the three transitions to account for "shutting down" the threads:

(5) the last transition of No after Yes has finished: $(1, 0, 0, 1) \xrightarrow{\text{Yes}} (0, 1, 0, 1)$;
(6) Yes terminates: $(m, b, 1, ne) \xrightarrow{\text{Yes}} (m, b, 0, ne)$ if $m \leq 1$;
(7) No terminates: $(0, b, ye, 1) \xrightarrow{\text{No}} (0, b, ye, 0)$ if $m \leq 1$.

It is easy to check that the only transition which do not decrease the state, ordered with lexicographic order on (m, b) and the product order on $((m, b), ye, ne)$ are the two loop transitions. Moreover, in a state (m, b, ye, ne), all the transitions labeled with "if $b = 1$ then Yes else No" decrease the state. This means that the following criterion shows the model is fairly terminating.

E.4 Locally fairly terminating models

Definition E.2. A fairness model $\mathcal{F}$ is called *locally fairly terminating* if there exists a well founded order $\leq$ over $\mathcal{F}$ and a map $\pi : \mathcal{F} \to \mathcal{R}$ from states to roles which satisfies the following conditions:

(1) for all transitions $\delta \xrightarrow{\rho} \delta'$, $\delta' \leq \delta$;
(2) for all states $\delta \in \mathcal{F}$ which are not dead ends, $\pi(\delta) \in \mathit{enabled\_roles}\ \delta$ and for all δ' such that $\delta \xrightarrow{\pi(\delta)} \delta'$, $\delta' < \delta$;
(3) for all transitions $\delta \xrightarrow{\rho} \delta'$, if $\rho \neq \pi(\delta)$, then $\pi(\delta') = \pi(\delta)$;

where a state δ is called a dead end if there are no outgoing transitions from it.

We call this criterion *local* because it can be checked for each transition independently, without any reference to traces. This criterion is correct:

Lemma E.3. *If a fairness model $\mathcal{F}$ is locally fairly terminating, then it is fairly terminating.*

REFERENCES

Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*, Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle (Eds.). Lecture Notes in Computer Science, Vol. 60. Springer, 393–481. https://doi.org/10.1007/3-540-08755-9_9

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. 2020. Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. 336–365. https://doi.org/10.1007/978-3-030-44914-8_13

Amin Timany and Lars Birkedal. 2021. Reasoning about monotonicity in separation logic. In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*. 91–104. https://doi.org/10.1145/3437992.3439931

Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue (proof pearl). In *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*. 76–90. https://doi.org/10.1145/3437992.3439930