Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems

Technical Appendix

Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch,
Simon Oddershede Gregersen, and Lars Birkedal

February 5, 2020

A Helper Constructs

```
rec listenwait skt =
  match receivefrom skt with
    SOME m => m
  | NONE => listenwait skt
  end

rec listen skt handler =
  match receivefrom skt with
    SOME m => handle (π1 m) (π2 m)
  | NONE => listen skt handler
  end

rec listfold handler acc l =
  match l with
    SOME a =>
      let acc = handler acc (π1 a) in
      listfold handler acc (π2 a)
  | NONE => acc
  end
```

B Bag Service

In order to handle multiple client requests simultaneously servers may employ concurrency by forking multiple threads. However, such servers may still have data structures or resources that are not safe to use in a concurrent setting. It is therefore often necessary to deploy advanced synchronization mechanisms to ensure correctness. Fig. 1 shows the architecture of a concurrent bag service that exploits multiple threads in order to handle several client requests at the same time while working on a shared bag data structure.

Fig. 2 shows a thread-safe implementation of a bag module that uses a linked list as its internal representation of the bag and a lock in order to guarantee that only one thread at a time operates on the linked list. A weak, but still useful specification is the following: Given a predicate Ω, the bag contains elements x for which $\Omega(x)$ holds. When inserting an element we give away the resources,

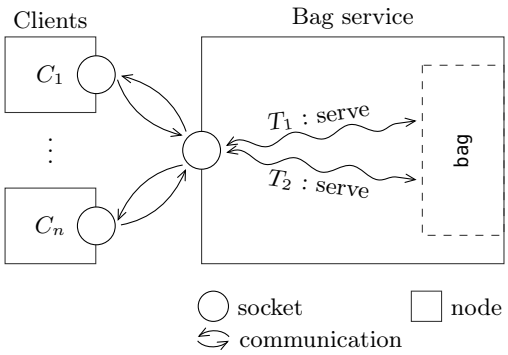


Fig. 1. The architecture of a concurrent bag service with threads working on a shared data structure governed by a lock and demonstrates Aneris' ability to support both shared-memory concurrency and distributed networking.

and when removing an element we give back an element plus the knowledge that it satisfies the predicate. This looks as follows:

$$
\begin{aligned}
&\exists\, \mathrm{isBag}\,.\\
&\quad\wedge\quad \forall n, v, \Omega.\ \mathrm{isBag}(n, v, \Omega) \dashv\vdash \mathrm{isBag}(n, v, \Omega) * \mathrm{isBag}(n, v, \Omega)\\
&\quad\wedge\quad \forall n, \Omega.\, \{\mathsf{IsNode}(n)\}\ \texttt{newbag}\ ()\, \{v.\ \mathrm{isBag}(n, v, \Omega)\}\\
&\quad\wedge\quad \forall v, e.\, \{\mathrm{isBag}(n, v, \Omega) * \Omega(e)\}\ \texttt{insert}\ v\ e\, \{\mathsf{True}\}\\
&\quad\wedge\quad \forall n, v, \Omega.\, \{\mathrm{isBag}(n, v, \Omega)\}\ \texttt{remove}\ b\, \{v. v = \mathsf{None} \vee \exists x.\, v = \mathsf{Some}\ x \wedge \Omega(x)\}
\end{aligned}
$$

Note how the isBag predicate is duplicable and therefore sharable among multiple threads. The isBag predicate is defined as follows:

$$
\begin{aligned}
P_{bag} &\triangleq \exists u.\, \ell \mapsto_n u * \mathrm{bagList}(\Omega, u)\\
\mathrm{isBag}(n, v, \Omega) &\triangleq \exists \ell, l.\, v = (\ell, l) * \mathrm{isLock}(n, l, P_{bag})
\end{aligned}
$$

where bagList is defined by recursion as the unique predicate satisfying

$$
\mathrm{bagList}(\Omega, u) \triangleq u = \mathtt{None} \vee \exists x, r.\, u = \mathtt{Some}\,(x, r) * \Omega(x) * \mathrm{bagList}(\Omega, r).
$$

Note that the isLock predicate is parameterized by a user-defined resource P_{bag} that follows the key resource: when the lock is acquired, the resources described by P_{bag} are given and the resources have to be given back when the lock is released.

Fig. 3 shows an AnerisLang implementation of a concurrent bag service. The main function creates a socket and a bag and forks two threads each executing the the `serve` function. This function listens for incoming messages. If the input message is an empty string it tries to remove an element from the bag and, if any, it sends the element back to the client, otherwise an empty string. If the input message is nonempty it inserts the message into the bag and acknowledges

```
rec newbag () =
  let l = ref NONE in
  let lock = newLock () in (l, lock)

rec insert b e =
  let l = (π1 b) in
  let lock = (π2 b) in
  acquire lock;
  l ← SOME (e, !l)
  release lock;
  res
```

```
rec remove b =
  let l = (π1 b) in
  let lock = (π2 b) in
  acquire lock;
  let res =
    (match !l with
       SOME p => l ← (π2 p); SOME (π1 p)
     | NONE   => NONE
    end) in
  release lock;
  res
```

Fig. 2. A thread-safe bag implemented using a linked list and a lock.

```
rec bag_service a =
  let skt = socket () in
  let bag = newbag ()
  socketbind skt a;
  fork { serve skt bag };
  fork { serve skt bag }

rec bag_client arg a server =
  let skt = socket () in
  socketbind skt a;
  sendto skt arg server;
  π1 (listenwait skt)
```

```
rec serve skt bag =
  (rec loop () =
    match receivefrom skt with
      SOME m =>
        if (π1 m) = "" then
          let v = remove bag in
          match v with
            SOME e => sendto skt v (π2 m)
            NONE   => sendto skt "" (π2 m)
          end
        else
          insert bag (π1 m);
          sendto skt "" (π2 m)
    | NONE => ()
    end;
    loop #()) ()
```

Fig. 3. An implementation of a concurrent bag service in AnerisLang. The bag service forks multiple threads for concurrently processing requests.

with an empty string. A bag client simply sends an argument to a server and returns the response.

In order to provide a specification for the bag service we define the socket protocol Φ_{bag} that will govern the socket on which the service listens for requests. Similar to the thread-safe bag implementation, the socket protocol will also be parameterized by a predicate Ω.

$$\begin{aligned} \mathit{insert}(\Omega, \Psi, m) &\triangleq \mathsf{body}(m) \neq \texttt{""} * \Omega(\mathsf{body}(m)) * \\ &\qquad \forall m'.\, \mathsf{body}(m') = \texttt{""} \mathrel{-\!\!*} \Psi(m') \\ \mathit{remove}(\Omega, \Psi, m) &\triangleq \mathsf{body}(m) = \texttt{""} * \\ &\qquad \forall m'.\, (\mathsf{body}(m') = \texttt{""} \vee \Omega(\mathsf{body}(m')) \mathrel{-\!\!*} \Psi(m')) \\ \Phi_{bag}(\Omega)(m) &\triangleq \exists \Psi.\, \mathsf{from}(m) \Rightarrow \Psi * (\mathit{insert}(\Omega, \Psi, m) \vee \mathit{remove}(\Omega, \Psi, m)) \end{aligned}$$

The protocol Φ_{bag} demands that the client should be bound to some protocol Ψ and that the server can receive two types of messages fulfilling either $\mathit{insert}(\Omega, \Psi, m)$ or $\mathit{remove}(\Omega, \Psi, m)$), corresponding to either inserting an element into the bag or removing one. To insert an element, the resources described by $\Omega(\mathsf{body}(m))$ has to be provided and it should suffice for the client to receive an empty string as a response. When asking to retrieve an element, either the answer is the empty string or the message will satisfy $\Omega(\mathsf{body}(m))$.

Using the socket protocol we can specify and verify the bag service as follows.

$$\begin{aligned} &\{\mathsf{Static}(a, A, \Phi_{bag}(\Omega)) * \mathsf{IsNode}(n)\} \\ &\quad \langle n; \texttt{bag\_service}\ a \rangle \\ &\{\mathsf{False}\} \end{aligned}$$

The client code can either add or remove an element from the bag service, and the specification is straightforward given a server address srv governed by $\Phi_{bag}(\Omega)$.

$$\begin{aligned} &\left\{ \begin{aligned} &\mathit{srv} \Rightarrow \Phi_{bag}(\Omega) * \mathsf{Dynamic}(a, A) * \mathsf{IsNode}(n) * \\ &\mathit{arg} = \texttt{""} \vee (\mathit{arg} \neq \texttt{""} \wedge \Omega(\mathit{arg})) \end{aligned} \right\} \\ &\quad \langle n; \texttt{bag\_client}\ \mathit{arg}\ a\ \mathit{srv} \rangle \\ &\{v.v = \texttt{""} \vee \Omega(v)\} \end{aligned}$$

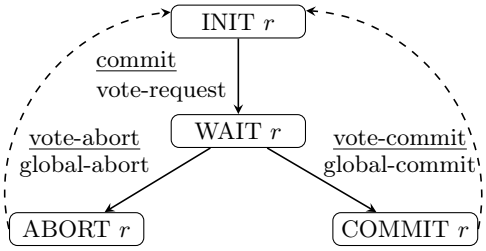
C Two-Phase Commit

Implementation. The two-phase commit protocol consists of the following two phases, each involving two steps:

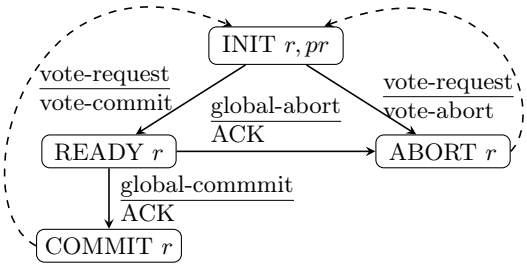
1. (a) The coordinator sends out a vote request to each participant.
   (b) A participant that receives a vote request replies with a vote for either commit or abort.
2. (a) The coordinator collects all votes and determines a result. If all participants voted commit, the coordinator sends a global commit to all. Otherwise, the coordinator sends a global abort to all.

(b) All participants that voted for a commit wait for the final verdict from the coordinator. If the participant receives a global commit it locally commits the transaction, otherwise the transaction is locally aborted. All participants must acknowledge.

These steps are shown as transition systems in Fig. 4 and an implementation of a TPC module that satisfies the conceptual description is shown in Fig. 5. Our abstract model differs slightly from the traditional diagram as we reuse the same code and sockets for communication between coordinators and participants. Every state is therefore tagged with a unique round number and dashed arrows are local transitions allowing reuse of the state transition systems by incrementing round numbers. To allow each participant to locally transition to the INIT state upon round completion and still communicating commit or abort, the INIT state is tagged with the previous result *pr* (initially, COMMIT suffices).



(a) The coordinator state transition system.



(b) The participant state transition system.

Fig. 4. Transition systems for the two-phase commit protocol.

The `tpc_coordinate` module expects an initial request message to be provided, along with a bound socket, a list of participants, and a function to make a decision when all votes have been received. Internally, it uses two local references; one to collect all the votes and one to count the number of acknowledgments.

The `tpc_participant` module expects a socket and two handlers—one to decide on a vote and one to finalize the decision made by the coordinator. When

invoked, the module listens for incoming requests, decides on a vote and waits for a global decision from the coordinator. Since each node can employ concurrency, the blocking wait for the decision does not prevent the client from doing concurrent work, in particular engaging in other rounds of TPC with other coordinators. Notice as well that there are no round numbers in the implementation; the round numbers are only in the abstract model to strengthen the specification.

```
rec tpc_coordinate m skt ps dec =
 let count = list_length ps in
 let msgs = ref (list_make ()) in
 let ack = ref 0 in
 list_iter (λn. sendto skt m n) ps;
 listen skt (rec handler m from =
   let msgs' = !msgs in
   msgs ← list_cons m msgs';
   if (list_length !msgs) = count
   then () else listen skt handler);
 let res = dec !msgs in
 list_iter (λn. sendto skt res n) ps;
 listen skt (rec h m from =
   ack ← !ack + 1;
   if !ack = count
   then res
   else listen skt h)
```

```
rec tpc_participant skt vote fin =
 let msg = listenwait skt in
 let act = vote (fst msg) in
 sendto skt act (snd msg);
 let res = listenwait skt in
 fin (fst res);
 sendto skt "ACK" (snd res);
 tpc_participant skt req fin
```

Fig. 5. An implementation of the two-phase commit protocol in AnerisLang. The functions `list_make`, `list_cons` and `list_length` are library utility functions for operations on lists implemented as splines.

Specification and Protocols. In order to specify and prove the TPC protocol correct, we will use the following resources, having a coordinator c and participants $p \in ps$:

- $\mathsf{Parts}(ps)$: Accounts for the set ps of participants for a concrete TPC round. The resource is duplicable and unmodifiable.
- $\mathsf{Coord}(p, r, s_c)$: Accounts for participant p's view of the coordinators current state s_C (*cf.* Fig. 4) in round r. The coordinator c owns an assertion regarding its own state $\mathsf{Coord}(c, r, s_C)$. We require that all parties agree which round and state the coordinator is in. Technically, this is stated in an invariant, I_{TPC}.
- $\mathsf{Part}(\pi, p, r, s_P)$: Accounts with fraction π for participant p's current state s_P (*cf.* Fig. 4) in round r.

To provide general, reusable implementations and specifications of the coordinator and participants implementing TPC, we do not define how requests, votes, nor decisions look like. We leave it to a user of the module to provide decidable predicates *isReq*, *isVote*, *isAbort* and *isGlobal* of type $(\mathit{String} \times \mathbb{N}) \to \mathsf{Prop}$. The user is free to pick $P : (\mathit{Address} \times \mathit{String}) \to \mathit{iProp}$ and $Q : (\mathit{Address} \times \mathbb{N}) \to \mathit{iProp}$,

the local pre- and postcondition for each participant. The socket protocol for the coordinator is shown below.

$$
\begin{aligned}
\Phi_{vote}(m) \triangleq\ & \exists p, r, ps.\, \mathsf{from}(m) = p * \mathsf{Parts}(\{p\} \cup ps)\ * \\
& \mathit{isVote}(\mathsf{body}(m), r) * \mathsf{Coord}(p, r, \textsc{wait}) * \\
& (\mathit{isAbort}(\mathsf{body}(m), r) * \mathsf{Part}(\tfrac{3}{4}, p, r, \textsc{abort}) \lor \\
& \neg \mathit{isAbort}(\mathsf{body}(m), r) * \mathsf{Part}(\tfrac{3}{4}, p, r, \textsc{ready})) \\
\Phi_{ack}(m) \triangleq\ & \exists p, r, ps, m', cs, pr.\, \mathsf{from}(m) = p * \mathsf{Parts}(\{p\} \cup ps)\ * \\
& \mathsf{Part}(\tfrac{3}{4}, p, r, \textsc{init}\ pr) * \\
& (\mathsf{Coord}(p, r, \textsc{commit}) * pr = \textsc{commit} * Q(p, r) \lor \\
& \mathsf{Coord}(p, r, \textsc{abort}) * pr = \textsc{ready} * P(p, m')) \\
\Phi_{coord}(m) \triangleq\ & \Phi_{vote}(m) \lor \Phi_{ack}(m)
\end{aligned}
$$

For a participant p to send a vote to the coordinator c, it has to show that it is indeed a participant $\mathsf{Parts}(\{p\} \cup ps)$, that the message is a vote for that round, that it knows the coordinator is in the WAIT state, $\mathsf{Coord}(p, r, \textsc{wait})$, and that the logical state of p matches p's actual vote. For the participant to send an acknowledgment, it has to prove it transitioned to the INIT pr where pr should match the global decision made by the coordinator. If the decision was to commit, the participant provides the updated resources for Q, otherwise it returns the resources described by P.

The socket protocol for the participants is as follows:

$$
\begin{aligned}
\Phi_{req}(p)(m) \triangleq\ & \exists r, ps, s_P.\, \mathsf{Parts}(\{p\} \cup ps) * P(\mathsf{body}(m), p) * \\
& \mathit{isReq}(\mathsf{body}(m), r + 1) * \mathsf{from}(m) \Rrightarrow \Phi_{coord} * \\
& \mathsf{Part}(\tfrac{3}{4}, p, r, \textsc{init}\ s_P) * \mathsf{Coord}(p, r + 1, \textsc{wait}) \\
\Phi_{glob}(p)(m) \triangleq\ & \exists r, ps, ga, ms, s_C, s_P.\, \mathsf{Parts}(\{\mathsf{from}(m') \mid m' \in ms\}) * \\
& \mathit{isGlobal}(\mathsf{body}(m), r) * \mathsf{from}(m) \Rrightarrow \Phi_{coord} * \\
& \mathsf{Part}(\tfrac{3}{4}, p, r, s_P) * \mathsf{Coord}(p, r, s_C) * \\
& ga = \{m' \mid m \in ms \land \mathit{isAbort}(m', r)\} * \\
& \left(\mathop{\Large *}_{m' \in ms} \mathit{isVote}(\mathsf{body}(ms), r)\ * \mathsf{R}(m') \right) * \\
& (ga = \emptyset \land \neg \mathit{isAbort}(\mathsf{body}(m), r) \land s_C = \textsc{commit}) \lor \\
& (ga \neq \emptyset \land \mathit{isAbort}(\mathsf{body}(m), r) \land s_C = \textsc{abort}) \\
\Phi_{part}(p)(m) \triangleq\ & \Phi_{req}(p)(m) \lor \Phi_{glob}(p)(m)
\end{aligned}
$$

In order to send a request for a round $r + 1$ of TPC to a participant p, a coordinator has to show p is indeed a participant of this instance and provide the resource described by P. The request should also be valid (through the *isReq* predicate) and the coordinator should be bound to the coordinator protocol Φ_{coord}. Furthermore, the coordinator has to show it is in the WAIT state and give up $\mathsf{Part}(\frac{3}{4}, p, r, \textsc{init}\ s_P)$ in order to allow the participant to make a transition.

The coordinator can broadcast a global decision when having received a message from all the participants (where ms is the set of messages received) and the decision is a valid global decision. All the messages has to have been received and be valid votes ($\ast_{m' \in ms} isVote(\mathsf{body}(ms), r) * \mathsf{R}(m')$). The coordinator also has to be honest: if any participant replied with an abort message ($ga \neq \emptyset$), the global message and the final state of the coordinator has to be ABORT.

Notice that for each message to a participant, the coordinator will provide the assertion $\mathsf{from}(m) \Rightarrow \Phi_{coord}$. This means the coordinator do not have to be primordial since the participant does not need to have prior knowledge of the coordinator. The coordinator could change from round to round.

With the TPC protocols in place, we can finally give a specification to the two TPC modules. The `tpc_participant` specification is straightforward:

$$\begin{array}{l}\left\{\begin{array}{l} I_{TPC} * isReqSpec(req) * isFinSpec(fin) * \mathsf{Parts}(ps) * \\ z \hookrightarrow_n \mathsf{Some}\ p * p \Rightarrow \Phi_{part}(p) * \mathsf{Part}(\frac{1}{4}, p, r, \mathsf{INIT}\ s_P) \end{array}\right\} \\ \quad \langle n; \texttt{tpc\_participant}\ z\ req\ fin \rangle \\ \{\mathsf{True}\}\end{array}$$

where req and fin are appropriate handlers for requests and finalization. The specification requires ownership of a bound socket bound by the participant protocol $\Phi_{part}(p)$ and fractional ownership of its own state, initialized to be INIT. Furthermore, the handlers req and fin should satisfy simple specifications that we elide to the Coq development.

The specification for `tpc_coordinate` is more involved:

$$\begin{array}{l}\left\{\begin{array}{l} I_{TPC} * isDecSpec(dec) * isReq(m, r+1) * \mathsf{Parts}(ps) * \mathsf{IsNode}(n) \\ z \hookrightarrow_n \mathsf{Some}\ c * a \Rightarrow \Phi_{coord} * \mathsf{Coord}(c, r, \mathsf{INIT}\ s_C) \\ \left(\ast_{p \in ps}\ p \Rightarrow \Phi_{part}(p) * \mathsf{Part}(\frac{3}{4}, p, r, \mathsf{INIT}\ s_P) * \mathsf{Coord}(p, r, \mathsf{INIT}\ s_C) * P(p, m)\right) \end{array}\right\} \\ \quad \langle n; \texttt{tpc\_coordinate}\ m\ z\ ps\ dec \rangle \\ \left\{\begin{array}{l} \langle n; v \rangle.\ \exists s_C, s_P.\ isGlobal(v, r+1) * \mathsf{Coord}(c, r+1, s_C) * z \hookrightarrow_n \mathsf{Some}\ c \\ \qquad \left(\ast_{p \in ps}\ \mathsf{Coord}(p, r+1, s_C) * \mathsf{Part}(\frac{3}{4}, p, r, \mathsf{INIT}\ s_P)\right) * \\ \qquad \left(isAbort(v, r+1) * s_C = \mathsf{ABORT} * s_P = \mathsf{ABORT} * \ast_{p \in ps}\ \exists m.\ P(p, m)\right) \vee \\ \qquad \left(\neg isAbort(v, r+1) * s_C = \mathsf{COMMIT}\ * s_P = \mathsf{COMMIT} * \ast_{p \in ps}\ Q(p, r+1)\right) \end{array}\right\}\end{array}$$

To invoke `tpc_coordinate`, one has to provide a valid request m, a socket z already bound to some address guarded by the Φ_{coord} protocol, a list of participants p, and a decision handler dec. For each participant p, the address should be governed $\Phi_{part}(p)$ and the resources describing the participant's view of its own and the

coordinator's state should be passed along. Finally, the resources described by $P(p, m)$ must also be provided.

The postcondition here is the most exciting part: it is exactly what one would expect. Either all participants along with the coordinator agreed to commit in which case we obtain $Q(p, r)$ for each participant p or they all agreed to abort, in which case we get back $P(p, m)$ for each participant p.

D A Replicated Log

We have implemented and verified a replicated logging system implemented on top of the TPC coordinator and participant modules to showcase vertical composition of complex protocols in Aneris.

An implementation of a replicated logging system is shown in Fig. 6. `logger` creates a socket `skt`, binds the address `a` to it, and initiates a TPC round for all databases in `dbs`. The decision handler `dec` is called by the TPC coordinator module when all votes have been received.

From the perspective of the database, `db`, an internal reference `log` keeps the log.[1] Upon an incoming request, the message is parsed and the proposed change is stored in the `wait` reference. If the global decision by `logger` is to commit, the string stored in `wait` will be appended to the log. To give a logical account

```
rec logger log a m dbs =
  let skt = socket () in
  let dec = λ msgs =
    let r = listfold (λ a, m . a && m = "COMMIT") true msgs in
    if r then "COMMIT" else "ABORT" in
  socketbind skt a;
  tpc_coordinate ("REQUEST_" ^ m) skt dbs dec
rec db addr =
  let skt = socket() in
  let wait = ref "" in
  let log = ref "" in
  let req = λ m . wait ← valof m; "COMMIT" in
  let fin = λ m .
    if m = "COMMIT"
    then log ← !log ^ !wait else () in
  socketbind skt addr;
  tpc_participant skt req fin
```

Fig. 6. An implementation in AnerisLang of a replicated logging system that uses the two-phase commit modules . ^ denotes string concatenation in AnerisLang.

of the local state of each database we introduce the fractional ghost resources $\mathsf{LOG}(\pi, p, l)$ and $\mathsf{PEND}(\pi, p, (l, s))$ that keep track of the log l and the proposed change s for each participant p. The predicates P and Q, which we instantiate

[1] Ideally, this would be stable storage, however, for the sake of the example a reference suffices.

TPC with, are defined below:

$$P_{rep}(p)(m) \triangleq \exists l, s.\, (m = \texttt{"REQUEST\_"} \mathbin{@} s) * \mathsf{LOG}(\tfrac{1}{2}, p, l) * \mathsf{PEND}(\tfrac{1}{2}, p, (l, s))$$
$$Q_{rep}(p)(n) \triangleq \exists l, s.\, \mathsf{LOG}(\tfrac{1}{2}, p, l@s) * \mathsf{PEND}(\tfrac{1}{2}, p, (l, s))$$

With the resources in place, the specification of `db` is straightforward and follows from the specification of the TPC participant module:

$$\left\{\begin{array}{l} I_{TPC} * \mathsf{Dynamic}(a, \Phi_{part}(a)) * \mathsf{IsNode}(n) * \\ \mathsf{Part}(\tfrac{1}{4}, a, r, \mathsf{INIT}\ s_P) * \mathsf{LOG}(\tfrac{1}{2}, a, \texttt{""}) \end{array}\right\}$$
$$\langle n; \texttt{db}\ a \rangle$$
$$\{\mathsf{True}\}$$

$$\left\{\begin{array}{l} I_{TPC} * \mathsf{Parts}(dbs) * \mathsf{FreePort}(a) * isReq(m) * \mathsf{Part}(\tfrac{3}{4}, p, r, \mathsf{INIT}\ s_P) * \\ \mathop{\Large *}_{p \in dbs} \exists s_P, p \Rrightarrow \Phi_{part}(p) * \mathsf{Coord}(p, r, \mathsf{INIT}\ s_P) * P_{rep}(p, m) \end{array}\right\}$$
$$\langle n; \texttt{logger}\ log\ a\ m\ dbs \rangle$$
$$\left\{\begin{array}{l} \langle n; v \rangle.\, \exists m, r.\ \mathop{\Large *}_{p \in dbs} \exists s_P.\, \mathsf{Coord}(p, r, \mathsf{INIT}\ s_P) * \mathsf{Part}(\tfrac{3}{4}, p, r, \mathsf{INIT}\ s_P) * \\ \qquad \left(v = \texttt{"COMMIT"} * \mathop{\Large *}_{p \in dbs} Q_{rep}(p, r)\right) \vee \left(v = \texttt{"ABORT"} * \mathop{\Large *}_{p \in dbs} P_{rep}(p, m)\right) \end{array}\right\}$$

Verification of our replicated logging client using two-phased-commit follows directly in a modular, node-local fashion by applying the specification of `tpc_coordinate`. Due to the TPC specification, this implies that if the global decision was to commit a change this change will have happened locally on all databases, *cf.* $\mathsf{LOG}(\frac{1}{2}, p, l@s)$ in Q_{rep}, and if the decision was to abort, then the log remains unchanged on all databases, *cf.* $\mathsf{LOG}(\frac{1}{2}, p, l)$ in P_{rep}.