Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols

LÉON GONDELMAN, Aarhus University, Denmark
JONAS KASTBERG HINRICHSEN, Aarhus University, Denmark
MÁRIO PEREIRA, NOVA LINCS, NOVA School of Science and Technology, Portugal
AMIN TIMANY, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

We present a foundationally verified implementation of a reliable communication library for asynchronous client-server communication, and a stack of formally verified components on top thereof. Our library is implemented in an OCaml-like language on top of UDP and features characteristic traits of existing protocols, such as a simple handshaking protocol, bidirectional channels, and retransmission/acknowledgement mechanisms. We verify the library in the Aneris distributed separation logic using a novel proof pattern—dubbed the *session escrow pattern*—based on the existing escrow proof pattern and the so-called *dependent separation protocols*, which hitherto have only been used in a non-distributed concurrent setting. We demonstrate how our specification of the reliable communication library simplifies formal reasoning about applications, such as a *remote procedure call library*, which we in turn use to verify a *lazily replicated key-value store with leader-followers* and clients thereof. Our development is highly modular—each component is verified relative to specifications of the components it uses (not the implementation). All our results are formalized in the Coq proof assistant.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Separation logic**; **Hoare logic**; *Higher order logic*; **Program verification**; **Program specifications**.

Additional Key Words and Phrases: Distributed systems, separation logic, refinement, higher-order logic, concurrency, formal verification

ACM Reference Format:
Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. 2023. Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols. *Proc. ACM Program. Lang.* 7, ICFP, Article 217 (August 2023), 31 pages. https://doi.org/10.1145/3607859

1 INTRODUCTION

Distributed programming is in some respect similar to message-passing concurrency where threads coordinate through the exchange of messages. However, contrary to communication between threads, network communication is *unreliable* (messages can be dropped, reordered, or duplicated) and *asynchronous* (messages arrive with a delay, which, in the presence of network partitions, is in general indistinguishable from a connection loss, *e.g.*, due to a remote machine crash).

Implementations of distributed applications therefore often rely on a *transport layer*, such as TCP or SCTP, to provide reliable communication channels among network servers and clients.

Authors' addresses: Léon Gondelman, Aarhus University, Denmark, gondelman@cs.au.dk; Jonas Kastberg Hinrichsen, Aarhus University, Denmark, hinrichsen@cs.au.dk; Mário Pereira, NOVA LINCS, NOVA School of Science and Technology, Portugal, mjp.pereira@fct.unl.pt; Amin Timany, Aarhus University, Denmark, timany@cs.au.dk; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

2475-1421/2023/8-ART217
https://doi.org/10.1145/3607859

Here "reliable" refers to the requirement that a server must process client requests in the order they are issued (FIFO order) and should not process any request more than once.[1] Such transport layer libraries often share two common traits: (1) they provide a high-level API which hides the implementation details by means of which reliable communication is achieved, and (2) the API they provide is stated in terms of BSD (Berkeley Software Distribution) socket-like API primitives *connect, listen, accept, send*, and *recv* that allow establishing asynchronous client-server connections and to transmit data via bidirectional channels.

It is well-known that the implementation and use of a transport layer library is challenging and error-prone [Guo et al. 2013] and thus it is a good target for formal verification. In recent years there has been much research progress on tools for analysis and verification of distributed systems using various techniques, ranging from model checking to mechanised verification in proof assistants. However, most of this research is situated on one of two ends of a spectrum, regarding how the reliable communication (when it is required) is treated.

On one end, existing work focuses on high-level properties of distributed applications *assuming* reliability, *e.g.*, assuming that the underlying transport layer of the verification framework is (partially) reliable [Gondelman et al. 2021; Krogh-Jespersen et al. 2020; Sergey et al. 2018], or that the shim connecting the analysis framework to executable code is reliable [Lesani et al. 2016; Wilcox et al. 2015]. This approach can limit the verified guarantees and lead to discrepancies between the specification, verification tool, and shim of such verified distributed systems [Fonseca et al. 2017]. On the other end of the spectrum, existing work focuses on *verifying* reliability and correctness properties of *protocols* for reliable communication, *e.g.*, formalization of the TCP protocol implementations [Bishop et al. 2006; Smith 1996], sliding window protocol verification in μCRL [Badban et al. 2005], or Stenning's protocol verified in Isabelle [Compton 2005]. This line of wok does not capture the reliability guarantees in a logic in a modular way that facilitates reasoning about clients of those protocols.

The purpose of the work presented in this paper is to show how we can tie these two loose ends of the spectrum. Concretely, in this paper we present the first modularly specified and verified implementation of a reliable communication library (RCLib), verified on top of an unreliable network. Our specifications enable modular verification of full functional correctness properties of distributed applications implemented on top of the reliable communication library. In the rest of this section we discuss the implementation (Section 1.1), specifications (Section 1.2), and verification methodology (Section 1.3) of RCLib, and the examples we have verified on top of the library (Section 1.4). We conclude with a list of the concrete contributions made by the paper (Section 1.5).

1.1 Formally Verifiable Implementation of a Reliable Communication Library

Our implementation of the reliable communication library aims at a high level of realism by employing realistic features such as asynchronous asymmetric channel creation (using 4-way handshake *à la* SCTP) and uses standard techniques such as sequence identifiers, retransmissions/acknowledgments, and channel descriptors for bidirectional data transmission. The channel descriptors consist of a local physical state containing send and receive buffers that is mutated both by user calls to the send/receive operations and the internal protocol procedures (running as concurrent threads) that enable reliable data transmission. Using buffers allows us to implement the user layer in a network-agnostic way, which is thus identical for the client and server, thus simplifying verification. Section 4 covers those implementation aspects and how they are verified in more detail.

[1]Because of network asynchrony it is very difficult to achieve exactly-once processing [Fekete et al. 1993; Gray 1979; Halpern 1987]. See [Ivaki et al. 2018] for a detailed survey of reliability notions in distributed systems.

```
let client clt srv  =
  let s =
    mk_client_skt str_ser int_ser clt in
  let c = connect s srv in
  send c "Carpe";
  send c "Diem";
  let m1 = recv c in
  let m2 = recv c in
  assert (m1 = 5 && m2 = 4)
```

```
let rec serve_loop c' =
  let req = recv c' in
  send c' (strlen req); serve_loop c'
let rec accept_loop s' =
  let c' = fst (accept s') in
  fork serve_loop c'; accept_loop s'
let server srv =
  let s' = mk_server_skt int_ser str_ser srv
     in
  server_listen s'; accept_loop s'
```

Fig. 1. Example: server returning the length of incoming strings.

In Figure 1 we present a simple example of how RCLib can simplify implementations of distributed programs. The example consists of a server that returns the length of incoming strings, and a simple client that connects to and communicates with the server. The right-hand side of Figure 1 shows the server implementation. The server is initialised calling `mk_server_skt int_ser str_ser srv`, where `int_ser str_ser` are serializers and `srv` is the server address. The function then returns a new socket `s'`, which is then put into listening mode with `server_listen s'`. Once the server is initialised, it starts an "accept" loop. The `accept s'` operation blocks until a new client connects, after which a channel descriptor c' (along with the client address, which we throw away), used to communicate with the client, is returned. The server serves each client on separate threads using a "serve" loop. The serve loop receives incoming strings, computes their length, and sends the results back. The left-hand side of Figure 1 shows the code for a particular client. The client first allocates its socket handler using `mk_client_skt str_ser int_ser clt`, and then connects to the server using `connect s srv` with the servers address `srv`. The operation blocks until the server accepts the request, after which it returns the channel descriptor c on which it communicates with the server. The client then sends two consecutive messages "Carpe" and "Diem", and waits for the results `m1` and `m2`. Note that for the client's assertion `assert (m1 = 5 && m2 = 4)` to hold, the communication must be reliable (in particular that messages arrive in order, and are not duplicated).

To verify the implementation of RCLib we need a formal operational semantics for distributed systems, along with a mechanism for reasoning about the semantics. To this end we have implemented the library in AnerisLang; an OCaml-like programming language with network primitives for UDP-like sockets that is formally defined in the Coq proof assistant together with the Aneris program logic [Krogh-Jespersen et al. 2020], which can be used to reason about unreliable distributed systems written in AnerisLang. Aneris allows reasoning about so-called *unreliable spatial resource transfer*—safely transferring spatial resources over an unreliable network—using a variant of the escrow pattern [Kaiser et al. 2017]. In particular, the escrow pattern lets a party safely store spatial resources in an "escrow", which another party can then obtain. The evidence that resources have been stored can be freely duplicated and thus one can repeatedly attempt to relay the evidence in the presence of failures. To ensure that only the receiving party can obtain the resources once, the receiving party starts with a one-time-use receipt, which must be given up along with a copy of the evidence, to obtain the transferred resources from the escrow. We will further elaborate on how Aneris shares resources using a similar intuition to the escrow pattern in Section 2.2.

As part of this work, to verify the AnerisLang implementation of RCLib, we have developed a simple compiler that translates programs written in a subset of OCaml to AnerisLang, to (informally) tie the verified program to executable code. This approach is similar to one taken in prior work [Chajed et al. 2019]. Note that such a compiler does not give any formal guarantees about the executed OCaml code: in fact, no such guarantees are currently possible, as OCaml does not have a formal semantics. Nevertheless, the formal operational semantics of AnerisLang matches, by design, the informal, but commonly understood, semantics of the corresponding subset of OCaml.

We remark that the trusted computing base of our framework (only) comprises *(a)* the compiler from OCaml to AnerisLang, *(b)* the operational semantics of AnerisLang, and *(c)* the Coq proof assistant in which we formalize all of our results. Note that Aneris is not part of the trusted computing base as its adequacy (soundness) is proven in Coq. Finally, we note that we focus on the challenges of giving reliable specifications to endpoints communicating in an unreliable network — we leave practical optimizations and performance evaluation to future work.

1.2 Modular Specifications of RCLib using Dependent Separation Protocols

To enable reasoning about functional correctness of the RCLib clients and libraries built on top of RCLib, we prove Aneris program logic specifications for the RCLib API. The key ingredient of these specifications is that they capture reliability guarantees of the client-server communication using *dependent separation protocols* of the Actris framework [Hinrichsen et al. 2020, 2022]. The dependent separation protocols are session type-like protocols that allow so-called *reliable dependent resource transfer*, by specifying a sequence of obligations to send or receive messages. Each exchange is associated with logical binders, a physical value, and separation logic resources, to specify obligations that the sender has to guarantee, and the receiver can rely on. Notably, the protocols are *dependent*, meaning that each specified exchange can depend on the exchanges that were made before them. As an example, the session of the echo-server example above can be captured by the following dependent separation protocol:

$$\mathsf{echo\_prot} \triangleq \mu rec.\ \mathbf{!}\,(s : \mathsf{String})\,\langle s \rangle.\,\mathbf{?}\langle |s| \rangle.\ rec$$

where $(s : \mathsf{String})$ denote a quantified variable and $\langle s \rangle$ and $\langle |s| \rangle$ denote the messages first sent and then received, respectively. The protocol specifies (from the client's point of view) how the client must first send a string s to the server, and how the server then replies with the length of the string $|s|$. The protocol is recursive, by virtue of the μ-operator. Given a protocol *prot* that describes a session for a specific client-server communication, we further follow the Actris methodology by associating each channel descriptor c in an established session with a predicate $c \xrightarrow[ser]{ip} prot$ in the logic called a *channel descriptor resource*. Here *ser* describes how the values must be serialized, *ip* corresponds to the ip-address of the node that the channel descriptor belongs to, and *prot* describes the current state of the protocol. When a session is established between a client and the server, along with fresh channel descriptors, the client obtains the resource $c \xrightarrow[ser]{ip} prot$, and the server obtains the resource $c \xrightarrow[ser]{ip} \overline{prot}$. Here, $\overline{prot}$ denotes the dual of the protocol *prot*, which turns all sends into receives, and vice versa. This enforces that what is sent by one side is what the other side receives. The local states of the protocol then change on each side with every user's call to the send and receive operations. We cover the specification pattern formally in Section 3, which presents the specification of the RCLib API and shows how it is used to prove the example above.

1.3 Modular Verification of RCLib and the Session Escrow Pattern

The dependent separation protocols that we adopt have previously been applied to verify reliable communication implemented via shared-memory message-passing [Hinrichsen et al. 2020, 2022]. In this paper, we apply the methodology to specify an implementation of reliable communication that is built on top of *unreliable* network primitives. This mismatch between a reliable proof pattern and an unreliable implementation imposes multiple non-trivial challenges, which we elaborate on in Section 2.4. Our solution to these challenges is a new proof pattern called the *session escrow pattern*, which is the main technical contribution of this paper. The session escrow pattern merges the *unreliable spatial resource transfer* of the escrow pattern (Section 1.1) with the *reliable dependent resource transfer* of Actris (Section 1.2), to achieve so-called *unreliable dependent resource transfer*.

In particular, the escrow pattern, and the Aneris variant thereof, allow us to transfer spatial resources in an unreliable setting, however, it is not suited for transfer of dependent resources in the same way as Actris. Consider the introductory example shown in Figure 1. The escrow of the escrow pattern would somehow need to capture the dependent history of prior messages, and similarly the witnesses would need to be applied sequentially, in the right order. The session escrow pattern thus serves as a solution to this problem. We present the session escrow pattern in Section 4.2, and discuss how it elegantly solves the non-trivial verification challenges.

1.4 Verified Reliable Distributed Components on top of the RCLib

We demonstrate the expressivity of the RCLib specifications by verifying a number of examples, including the simple example presented in Figure 1. As a more realistic case study, we implement a remote procedure call (RPC) service library. We additionally use the RPC library to implement and verify a *lazily replicated key-value store with leader-followers* implementation in which the leader can both read from and write to the contents of the store, and the followers lazily replicate the updates from the leader, preserving the order of the leader's writes.

We leverage the fact that Aneris allows us to obtain highly modular and general specifications. Indeed, each component (RCLib, RPC, leader-followers KVS) is verified relative to the *specification* of the libraries that it is built on top of (not their implementations); this simplifies reasoning since the specifications of libraries hide all the verification related details. For instance, the leader-followers is verified on top of the specification of the RPC library, which is expressed in terms of an abstract specification of the remote procedure calls. In particular, the verification of the leader-followers KVS does not involve any reasoning about network-level communication at all.

1.5 Contributions

In summary, the main contributions of this work are:

- RCLib: The first foundationally verified implementation of a reliable communication library for client-server communication with session-based protocol specifications (Section 3).
- The *session escrow pattern*; a proof pattern for reasoning about reliable dependent transfer of resources in an unreliable setting, used to verify the RCLib (Section 4).
- A demonstration of the expressivity of the RCLib specifications through the verification of a generic *remote procedure call* library, which can be used as a middleware component to further simplify the formal development of distributed applications (Section 5).
- A verified implementation of a leader-followers key-value store on top of the RPC library, demonstrating the vertical modularity enabled by the RCLib specifications (Section 6).

All of our results are mechanized on top of the Aneris logic and Actris framework in the Coq proof assistant, and consists of ~15.500 lines of Coq code. The development is available in the accompanying artifact [Gondelman et al. 2023].

2 PRIOR WORK AND ITS LIMITATIONS

In this section we recall some of the key features of the existing logical frameworks that our work builds on. We first give a brief overview of the Iris base logic followed by a short introduction of the Aneris program logic (Section 2.2) which is based on it. We then present the Actris ghost theory (Section 2.3) and subsequently discuss some of the limitations of Aneris and Actris (Section 2.4), motivating the construction of the *session escrow pattern*, which is presented in (Section 4).

2.1 An Iris Primer

The Iris base logic upon which Aneris is built is essentially a higher-order logic ($\land$, $\Rightarrow$, $\forall$, *etc.*) together with basic separation logic connectives, the separating conjunction, $*$ and the (magic) wand $-\!\!*$ along with user-defined ghost state and invariants:

$$P \in \mathsf{iProp} ::= \mathsf{True} \mid \mathsf{False} \mid P \Rightarrow P \mid P \land P \mid P \lor P \mid \forall x.\ P \mid \exists x.\ P \mid P * P \mid P -\!\!* P \mid \ \square P \mid \Rrightarrow P \mid \boxed{P}$$

The separating conjunction $P * Q$ asserts that the two propositions P and Q both hold, but, crucially, for *disjoint* resources. A prime example of a proposition defined in terms of ownership of resources is the heap points-to proposition $\ell \overset{ip}{\mapsto} v$ which asserts *exclusive* ownership over location ℓ, *i.e.* that it is allocated on the heap of the machine with IP address ip storing value v. The exclusiveness of heap points-to propositions is captured in the logic by the fact that $\ell \overset{ip}{\mapsto} v * \ell \overset{ip}{\mapsto} v' \vdash \mathsf{False}$ holds which means that location ℓ on machine ip cannot be owned twice as two disjoint resources. The opposite of exclusivity is persistence. In Iris the proposition $\square P$ ($\square$ being the so-called persistently modality) holds if P holds and asserts no exclusive ownership. Persistent propositions (P is called persistent if $P \dashv\vdash \square P$ — $\dashv\vdash$ being the logical equivalence relation) are duplicable, *i.e.*, $\square P \dashv\vdash \square P * \square P$. The wand connective has the same relation to the separating conjunction as the implication connective to ordinary conjunction, *i.e.*, $P \vdash Q -\!\!* R$ holds if and only if $P * Q \vdash R$. In addition to heap resources, Iris's base logic also supports user-declared ghost resources. We will not delve into the details of user-defined ghost resources in this paper. What is important here is the proposition $\Rrightarrow P$ where $\Rrightarrow$ is called the *update* modality. Intuitively, $\Rrightarrow P$ holds if we can update resources, including those asserted in invariants (see blow), in a consistent manner, *i.e.* without violating resources owned disjointly, *e.g.* by other threads or nodes, in order to satisfy P. We write $P \Rrightarrow Q$ as a shorthand for $\square(P -\!\!* \Rrightarrow Q)$ — an update that does not rely on any exclusive resources and is hence always valid to apply at any point (multiple times) in the proof. The proposition $\boxed{P}$ asserts that P holds invariantly. That is, once established, it must always hold throughout the program which means that we can rely on it holding before every individual step of execution but must ensure that it holds afterwards. Crucially, $\boxed{P}$ does not assert any exclusive ownership regardless of P. It just asserts that P must always hold — hence invariants are always persistent. We write Prop for the collection of Iris propositions as opposed to Prop, the collection of meta-level, *i.e.*, Coq, propositions.

2.2 Aneris: Distributed Separation Logic

Aneris is a program logic on top of Iris's base logic for reasoning about distributed programs in which each node is written in an ML-like language, with low-level socket-based communication primitives akin to the UDP protocol. An overview of the relevant additions can be found in Figure 2.

Specifications in Aneris are stated as ip-decorated Hoare triples $\{P\}\ \langle ip; e\rangle\ \{\Phi\}$. Aneris Hoare-triples capture partial correctness, *i.e.*, that given the precondition ($P : \mathsf{iProp}$), the program e can safely be executed *at the specified ip*, and as a result the postcondition ($\Phi : \mathsf{Val} \to \mathsf{iProp}$) holds for the returned value. We often write $\{P\}\ \langle ip; e\rangle\ \{w.Q\} \triangleq \{P\}\ \langle ip; e\rangle\ \{\lambda w.\, Q\}$, and $\{P\}\ \langle ip; e\rangle\ \{Q\} \triangleq \{P\}\ \langle ip; e\rangle\ \{\lambda w.\, w = () * Q\}$. Aneris's proof rules reflect the conventional informal specification of UDP-based communication, decorated with user-specified protocols for transfer of spatial resources of fresh messages, using an intuition similar to the escrow pattern which we discuss below. To express the proof rules Aneris employs new logical predicates that each govern parts of the network resources. The $\mathsf{FreeIp}(ip)$ resource[2] asserts that no node exists at the specified ip (ip). The $\mathsf{FreePorts}(ip, \mathcal{P})$ resource asserts that the specified ports ($\mathcal{P}$), at the specified ip (ip), have not been bound. We often write $\mathsf{FreeAddr}(sa) \triangleq \mathsf{FreePorts}(sa.\mathtt{ip}, \{sa.\mathtt{port}\})$. The $sh \overset{ip}{\hookrightarrow} (o, b)$ resource asserts exclusive ownership of a socket handler (sh) at the node with the given ip (ip), asserting

[2]Since Aneris is a resourceful separation logic, we often refer to predicates as resources.

Grammar:

$$\tau, \sigma ::= \text{Handle} \mid \text{Socket} \mid \text{Address} \mid \text{Message} \mid \ldots$$

$$P, Q ::= \text{FreeIp}(ip) \mid \text{FreePorts}(ip, \mathcal{P}) \mid sh \xhookrightarrow{ip} (o, b) \mid \text{Unallocated}(A) \mid sa \Mapsto \Phi \mid sa \rightsquigarrow (R, T) \mid \ldots$$

Network primitives:

Ht-start

$$\frac{\{P * \text{FreePorts}(ip, \mathcal{P})\}\ \langle ip;\ e\rangle\ \{w.\,\text{True}\}}{\{P * \text{FreeIp}(ip)\}\ \langle ip_{\text{sys}};\ \texttt{start}\ ip\ e\rangle\ \{w.\, w = ()\}}$$

Ht-newsocket

$$\{\text{True}\}\ \langle ip;\ \texttt{socket}\ ()\rangle\ \{w.\,\exists sh.\, w = sh * sh \xhookrightarrow{ip} (\text{None}, ✋)\}$$

Ht-settimeout

$$\{sh \xhookrightarrow{ip} (o, b)\}\ \langle ip;\ \texttt{settimeout}\ sh\ n\ m\rangle\ \{\texttt{if}\ (n = 0 \land m = 0)\ \texttt{then}\ sh \xhookrightarrow{ip} (o, ✋)\ \texttt{else}\ sh \xhookrightarrow{ip} (o, ⊘)\}$$

Ht-socketbind

$$\{\text{FreeAddr}(sa) * sh \xhookrightarrow{sa.\text{ip}} (\text{None}, b)\}\ \langle sa.\text{ip};\ \texttt{socketbind}\ sh\ sa\rangle\ \{w.\, w = 0 * sh \xhookrightarrow{sa.\text{ip}} (\text{Some}\ sa, b)\}$$

Ht-socket-interp-alloc

$$\frac{\{P * sa \Mapsto \Phi\}\ \langle ip;\ e\rangle\ \{w.\, Q\}}{\{P * \text{Unallocated}(\{sa\})\}\ \langle ip;\ e\rangle\ \{w.\, Q\}}$$

Ht-send

$$\left\{\begin{array}{l} sh \xhookrightarrow{m.\text{src.ip}} (\text{Some}\ m.\text{src}, b) * m.\text{dst} \Mapsto \Phi\ * \\ m.\text{src} \rightsquigarrow (R, T) * (m \notin T \Rightarrow \Phi\ m) \end{array}\right\}$$
$$\langle m.\text{src.ip};\ \texttt{sendto}\ sh\ m.\text{str}\ m.\text{dst}\rangle$$
$$\left\{\begin{array}{l} w.\, w = |m.\text{str}| * m.\text{src} \rightsquigarrow (R, T \cup \{m\})\ * \\ \quad sh \xhookrightarrow{m.\text{src.ip}} (\text{Some}\ m.\text{src}, b) \end{array}\right\}$$

Ht-recv

$$\{sh \xhookrightarrow{sa.\text{ip}} (\text{Some}\ sa, b) * sa \rightsquigarrow (R, T) * sa \Mapsto \Phi\}$$
$$\langle sa.\text{ip};\ \texttt{receivefrom}\ sh\rangle$$
$$\left\{\begin{array}{l} w.\, sh \xhookrightarrow{sa.\text{ip}} (\text{Some}\ sa, b)\ * \\ \quad (b = ⊘ * w = \texttt{None} * sa \rightsquigarrow (R, T)) \lor \\ \quad (\exists m.\, w = \texttt{Some}\ (m.\text{str}, m.\text{src}) * m.\text{dst} = sa\ * \\ \qquad sa \rightsquigarrow (R \cup \{m\}, T) * (m \notin R \Rightarrow \Phi\ m)) \end{array}\right\}$$

Fig. 2. The grammar and a selection of rules of the Aneris communication layer.

that it is either closed or open on a given address (o : Option Address), as well as whether it is in a blocking or non-blocking state (b : ✋ + ⊘). The Unallocated(A) resource asserts that the given addresses (A : Set Address) are “unallocated”, meaning that no protocols have been assigned to them yet. Conversely, once “allocated”, the duplicable $sa \Mapsto \Phi$ resource asserts that the socket at the given address (sa), follows the given protocol (Φ : Message $\to$ iProp). The $sa \rightsquigarrow (R, T)$ resource asserts the history of messages received (R) and transmitted (T) at the given address (sa). Finally, FreePorts$(ip, \mathcal{P})$ and Unallocated(A) enjoy splitting; FreePorts$(ip, \mathcal{P}_1 \uplus \mathcal{P}_2) \dashv\vdash$ FreePorts$(ip, \mathcal{P}_1)$ $*$ FreePorts$(ip, \mathcal{P}_2)$ and Unallocated$(A_1 \uplus A_2) \dashv\vdash$ Unallocated$(A_1) *$ Unallocated(A_2).

The semantics of these connectives is made clear by the associated logical rules. The rule Ht-start states that one can start new nodes using `start` $ip\ e$ whenever the ip address is free (FreeIp(ip)), giving back the assertion that all the ports ($\mathcal{P}$) at the ip are free (FreePorts$(ip, \mathcal{P})$). Note that `start` $ip\ e$ can only be executed on the “system” node with the ip address: ip_{sys}. This is to reflect that the language does not have dynamic allocation of nodes, and instead only allows setting up the nodes from an initial node that can be thought of as an admin node. The rule Ht-newsocket specifies that the `socket` () expression allocates a new socket, returning the handle (sh), for which we obtain exclusive ownership $sh \xhookrightarrow{ip} (\text{None}, ✋)$, capturing that the socket is initially unbound (None), and blocking (✋). The blocking status of a socket can be updated with `settimeout` $sh\ n\ m$, as specified by the Ht-settimeout rule, which sets the blocking status based on the given timeout float number $n.m$. Since Aneris is time-insensitive, the timeout is treated as the binary blocking flag in the logic. Concretely, that means that in the operational semantics of Aneris, a call to `receivefrom` sh with a non-blocking handle and when there is nothing to be returned to the user, is evaluated to the `None` value, modelling that timeout has expired. In particular the role of the numerical value $n.m$ is only to set the status to non-blocking or back to blocking, depending on whether $n.m = 0$ or not. Such an approach is sound logically, when reasoning about safety properties.

Sockets are bound to addresses via `socketbind` $sh\ sa$, as specified by the Ht-socketbind rule, that updates the socket handle resource to the bound address $sh \overset{ip}{\hookrightarrow} (\mathsf{Some}\ sa, b)$. Protocols are bound to socket addresses using the Ht-socket-interp-alloc rule, which captures that an $\mathsf{Unallocated}(\{sa\})$ resource can be converted to an $sa \Mapsto \Phi$ resource at any given time. The rule Ht-send specifies how a message ($m.\mathsf{str}$) can be sent over a socket (sh) to a given destination address ($m.\mathsf{dst}$) using the `sendto` $sh\ m.\mathsf{str}\ m.\mathsf{dst}$ command. It requires the socket to be bound to the sender's address ($sh \overset{ip}{\hookrightarrow} (\mathsf{Some}\ m.\mathsf{src}, b)$), and the protocol of the destination to be known ($m.\mathsf{dst} \Mapsto \Phi$). It also requires the message history resource ($m.\mathsf{src} \rightsquigarrow (R, T)$), which allows to track whether the message to be sent is fresh or not. If the message is fresh ($m \notin T$) we must give up the resources specified by the protocol ($\Phi\ m$). Otherwise, *i.e.* resending a previously sent message is done without any resource transfer. In return, the sent message is added to the history of transmitted message ($T \cup \{m\}$).

The rule Ht-recv specifies how we can receive messages over a socket (sh) using `receivefrom` sh. This requires that the socket is bound ($sh \overset{ip}{\hookrightarrow} (\mathsf{Some}\ sa, b)$), the presence of the message history ($sa \rightsquigarrow (R, T)$), and that we know our protocol ($sa \Mapsto \Phi$). If there is no message inbound the function returns nothing ($w = \mathsf{None}$), and we retain the original history of received messages (R). If there is a message inbound its contents are returned ($w = \mathsf{Some}\ m.\mathsf{str}, m.\mathsf{src}$), and it is added to the history of received messages ($R \cup \{m\}$). Finally, if the message is fresh ($m \notin R$) we obtain the resources specified by our protocol ($\Phi\ m$). If the socket is blocking the function blocks until a message is received, and thus the first case is only possible if the socket is non-blocking ($b = ⊘$).

Adequacy: obtaining the initial resources and closed proofs. The specifications presented above depend on resources that seem to occur from nothing. Particularly $\mathsf{Unallocated}(A)$, $sa \rightsquigarrow (R, T)$, and $\mathsf{FreeIp}(ip)$ occur as a precondition of various rules, while not being obtained as a result of any. This is sound because closed proofs of complete programs in Aneris are instantiated with a concrete network configuration, for which initial resources are provided. This is formally captured by the foundationally mechanised Aneris adequacy theorem:

Theorem 2.1 (Adequacy of Aneris). *Let $\varphi \in \mathsf{Val} \rightarrow \mathsf{Prop}$ be a meta-level (i.e. Coq) predicate on values and suppose that the following is derivable in Aneris for a program e running on node ip:*

$$ip \notin ips \Rightarrow \left\{\mathsf{Unallocated}(A) * \mathop{\Huge *}_{a\in A}, a \rightsquigarrow (\emptyset, \emptyset) * \mathop{\Huge *}_{n\in ips}, \mathsf{FreeIp}(n)\right\} \langle ip;\ e\rangle\ \{\varphi\}$$

We then obtain the following properties:

- ***Safety:*** *The program e, i.e., all threads on all nodes, will never get stuck*
- ***Postcondition Validity:*** *If the program e terminates with value v, then $\varphi\ v$ holds.*

To verify a complete program, the first step is thus to pick the set of addresses (A : Set Address), and the set of ips (excluding the ip of the initial node) (ips : Set Ip where $ip \notin ips$); and then prove the Hoare triple, in which we start with the initial network resources for tracking unallocated addresses ($\mathsf{Unallocated}(A)$), socket histories ($*_{a\in A}, a \rightsquigarrow (\emptyset, \emptyset)$), and free ips ($*_{n\in ips}, \mathsf{FreeIp}(n)$).

To obtain a closed proof of a distributed system we apply the adequacy theorem to the admin node (with ip ip_{sys}). To establishing statically known protocols one can apply the Ht-socket-interp-alloc rule before starting the individual nodes, to obtain the duplicable $sa \Mapsto \Phi$ resources which can be distributed to the nodes when started.

2.3 Actris: Dependent Separation Protocols

The Actris framework [Hinrichsen et al. 2022] supports specifying and reasoning about reliable communication. It does so by using a notion of session-type-inspired separation logic protocols, called *dependent separation protocols*, defined by the following three constructors:

$$\mathit{prot} \in \mathsf{iProto} ::= \mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} \mid \mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} \mid \mathbf{end}$$

These constructors are used to specify a sequence of obligations to send (**!**) and receive (**?**), which can be terminated by **end**. More specifically, the constructors $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ and $\mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$ specify an exchange of a value v, along with resources described by P, given an instantiation of the binders $\vec{x}:\vec{\tau}$. The binders $\vec{x}:\vec{\tau}$ bind into both the value v, the proposition P, and the tail *prot*. The latter means that the protocols are *dependent*, i.e., that message exchanges can depend on the exchanges that were made before them. Additionally, dependent separation protocols can be defined recursively using the Aneris μ-operator (most of the protocols presented in this paper are recursive). Finally, we often write $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle.\,\mathit{prot}$ instead of $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{\mathsf{True}\}.\,\mathit{prot}$.

The dependent separation protocols are subject to the conventional session type notion of *duality* $\overline{\mathit{prot}}$, which turns all sends (**!**) into receives (**?**), and vice versa, for the given protocol *prot*:

$$\overline{\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}} = \mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\overline{\mathit{prot}} \qquad \overline{\mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}} = \mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\overline{\mathit{prot}} \qquad \overline{\mathbf{end}} = \mathbf{end}$$

By this notion of duality, we can guarantee that any two programs with dual protocols will have well-behaved communication by construction; when one endpoint expects some message and resources, the other endpoint will send just that, and vice versa.

As an example consider the following dependent separation protocol of a simple echo-server:

$$\mathsf{echo\_prot} \triangleq \mu\mathit{rec}.\ \mathbf{?}(s : \mathsf{String})\,\langle s\rangle.\,\mathbf{!}\,\langle |s|\rangle.\,\mathit{rec}$$

The protocol specifies (from the server's point of view) how the server first receives an arbitrary string s from the client. The server then replies with the length of the string $|s|$, and then recurses.

The dependent separation protocols enjoy a so-called *subprotocol* relation ($\sqsubseteq$), which captures *protocol-preserving updates*: local changes that are indistinguishable by the other party, and which are therefore safe to perform without coordination. The most prominent such protocol-preserving update is that of *swapping*, formally captured by the following relation:

$\sqsubseteq$-SWAP

$$\frac{(\vec{x} : \vec{\tau})\ \#\#\ (\vec{y} : \vec{\sigma})}{\mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathbf{!}\,\vec{y}:\vec{\sigma}\,\langle w\rangle\{Q\}.\,\mathit{prot} \sqsubseteq \mathbf{!}\,\vec{y}:\vec{\sigma}\,\langle w\rangle\{Q\}.\,\mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}}$$

The rule captures that one can choose to send (**!**), a message, before the prior receive (**?**), whenever their binders are disjoint (this condition ensures that the send is independent of the receive).

The subprotocol relation is akin to the notion of *subtyping*, in which supertypes carry less information/behaviours than their subtypes. As an example, a user of a protocol can choose to send a message earlier than required (by swapping it ahead of a receive), but not vice versa. Swapping the message ahead (by using the swapping rule) thus means that the protocol now permits less behaviours than before. In particular, we have $\mathbf{?}\langle 42\rangle.\,\mathbf{!}\,\langle \mathtt{true}\rangle.\,\mathbf{end} \sqsubseteq \mathbf{!}\,\langle \mathtt{true}\rangle.\,\mathbf{?}\langle 42\rangle.\,\mathbf{end}$, but *not* the inverse $\mathbf{!}\,\langle \mathtt{true}\rangle.\,\mathbf{?}\langle 42\rangle.\,\mathbf{end} \not\sqsubseteq \mathbf{?}\langle 42\rangle.\,\mathbf{!}\,\langle \mathtt{true}\rangle.\,\mathbf{end}$, as the first protocol permits sending both as the first and second action, while the second protocol only permits sending as the first action.

To see why the subprotocol relation is useful, consider a situation where a client of the echo-server sends two messages upfront, and only awaits the responses from the server afterwards. The protocol of such a client cannot possibly be *strictly* dual to the server's echo_prot protocol, and so it might seem that its communication with the server is not inherently sound. However, we

$$\text{True} \Rrightarrow \exists \chi.\, \text{prot\_ctx}\ \chi\ \epsilon\ \epsilon * \text{prot\_own}_{\text{l}}\ \chi\ \mathit{prot} * \text{prot\_own}_{\text{r}}\ \chi\ \overline{\mathit{prot}} \qquad \text{(PROTO-ALLOC)}$$

$$\begin{aligned}&\text{prot\_ctx}\ \chi\ \vec{v_1}\ \vec{v_2} * \text{prot\_own}_{\text{l}}\ \chi\ (!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) * P[\vec{t}/\vec{x}] \Rrightarrow\\ &\quad\left(\triangleright^{|\vec{v_2}|}\ \text{prot\_ctx}\ \chi\ (\vec{v_1}\cdot[v[\vec{t}/\vec{x}]])\ \vec{v_2}\right) * \text{prot\_own}_{\text{l}}\ \chi\ (\mathit{prot}[\vec{t}/\vec{x}])\end{aligned} \qquad \text{(PROTO-SEND-L)}$$

$$\begin{aligned}&\text{prot\_ctx}\ \chi\ \vec{v_1}\ ([w]\cdot\vec{v_2}) * \text{prot\_own}_{\text{l}}\ \chi\ (?\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) \Rrightarrow\\ &\quad\triangleright\,\exists\vec{y}.\,(w = v[\vec{y}/\vec{x}]) * P[\vec{y}/\vec{x}] * \text{prot\_ctx}\ \chi\ \vec{v_1}\ \vec{v_2} * \text{prot\_own}_{\text{l}}\ \chi\ \mathit{prot}[\vec{y}/\vec{x}]\end{aligned} \qquad \text{(PROTO-RECV-L)}$$

$$\text{prot\_own}_{\text{l}}\ \chi\ \mathit{prot} * (\mathit{prot} \sqsubseteq \mathit{prot}') \Rrightarrow \text{prot\_own}_{\text{l}}\ \chi\ \mathit{prot}' \qquad \text{(PROTO-}\sqsubseteq\text{-L)}$$

Fig. 3. Excerpt of rules of the Actris ghost theory.

can guarantee that it is sound, by updating the initially strictly dual protocol, using the sound subprotocol relation, so that the dual of the echo_prot fits the client:

$$\overline{\texttt{echo\_prot}} \sqsubseteq\, !\,(s_1 : \text{String})\,\langle s_1\rangle.\, !\,(s_2 : \text{String})\,\langle s_2\rangle.\, ?\langle|s_1|\rangle.\, ?\langle|s_2|\rangle.\, \overline{\texttt{echo\_prot}}$$

As the client's first receive and second send are independent, the relation follows directly from unfolding the recursive definition twice, and using the ⊑-SWAP rule (and omitted structural rules).

Actris ghost theory. The Actris framework includes a logical model of *reliable dependent resource transfer* via the dependent separation protocols called the *Actris ghost theory* which is shown in Figure 3. The model operates on three resources called prot_ctx $\chi\ \vec{v_1}\ \vec{v_2}$ and prot_own$_{\text{l}}$ χ *prot* and prot_own$_{\text{r}}$ $\chi\ \overline{\mathit{prot}}$. The prot_ctx $\chi\ \vec{v_1}\ \vec{v_2}$ resource acts as a shared context that tracks the messages that are currently in transit in either direction via $\vec{v_1}$ and $\vec{v_2}$ respectively; we refer to $\vec{v_1}$ and $\vec{v_2}$ as (reliable) buffers. The resources prot_own$_{\text{l}}$ χ *prot* and prot_own$_{\text{r}}$ $\chi\ \overline{\mathit{prot}}$ represent the current view of the session from the perspective of either endpoint. The resources are parameterised by an identifier χ that associate them with each other. The rules of the ghost theory capture how to allocate these resources (PROTO-ALLOC), how to release resources along with sent values (PROTO-SEND-L) and how to acquire resources along with received values (PROTO-RECV-L). We omit the symmetric rules about the transfer from right to left. The rules are defined in terms of the *viewshift* connective $P \Rrightarrow Q$, which intuitively captures that the ghost state described by P can safely be updated to the ghost state described by Q. This is made precise by the following rule:

$$\text{Vs-csq}\quad \frac{P_1 \Rrightarrow P_2 \qquad \{P_2\}\,\langle ip;\, e\rangle\,\{w.\,Q_2\} \qquad \forall w.\, Q_2 \Rrightarrow Q_1}{\{P_1\}\,\langle ip;\, e\rangle\,\{w.\,Q_1\}}$$

We use the viewshift and this rule when instantiating the ghost state of our reliable communication framework as presented in Section 3.

The PROTO-ALLOC rule captures that we can always allocate a new session with a fresh indentifier χ, and some freely picked protocol *prot*. The PROTO-SEND-L rule captures that to send a value, the protocol must be in a sending state (prot_own$_{\text{l}}$ $\chi\ !\,\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$). We must then provide a concrete instantiation ($\vec{t} : \vec{\tau}$) of the binders ($\vec{x}{:}\vec{\tau}$), and give up the resources ($P[\vec{t}/\vec{x}]$). As a result, we get back the shared context with the message (for the given binder instantiation) added at the end of the respective buffer (prot_ctx χ ($\vec{v_1}\cdot[v[\vec{t}/\vec{x}]])\vec{v_2}$). We also get back the protocol resource whose protocol is updated to its dependent tail ($\mathit{prot}[\vec{t}/\vec{x}]$). The PROTO-RECV-L rule specifies that the protocol must be in a receiving state (prot_own$_{\text{l}}$ χ $(?\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot})$), and that there is a message in the inbound buffer prot_ctx $\chi\ \vec{v_1}\ ([w]\cdot\vec{v_2})$. We then get an instantiation of the binders ($\vec{y} : \vec{\tau}$) as specified by the protocol ($\vec{x}{:}\vec{\tau}$), for which we obtain ownership of the resources specified by the protocol ($P[\vec{y}/\vec{x}]$). We additionally learn that the received value (w) is equal to the value of the protocol ($w = v[\vec{y}/\vec{x}]$). Finally, we get back the shared context with the message removed from the buffer (prot_ctx $\chi\ \vec{v}_1\ \vec{v}_2$) and the protocol resource whose protocol is updated to its dependent tail ($prot[\vec{t}/\vec{y}]$). The PROTO-$\sqsubseteq$-L rule specifies that we can update the local protocol resource according to the subprotocol relation $\sqsubseteq$. Note that the conclusions of the rules are guarded by the later modality $\triangleright$, and its iterated version $\triangleright^n$; this is due to the higher-order nature of the ghost theory.

2.4 Limitations of Prior Work

Aneris and Actris individually solve the problems of *unreliable spatial resource transfer* and *reliable dependent resource transfer*, respectively. However, neither of these are expressive enough for the verification of our framework, which ultimately relies on *unreliable dependent resource transfer.*

Aneris and the escrow pattern. As described in Section 1.1 Aneris mimics the intuition of the escrow pattern to achieve unreliable spatial resource transfer. In particular, the escrow pattern manifests in Aneris by treating the socket interpretation $sa \Mapsto \Phi$ as a replicated resource agreement between any external party and the socket address sa. The history of transmitted messages T, tracked by the message histories $sa \rightsquigarrow (R, T)$ then evidence that the resources have been put into the escrow, and therefore we can repeatedly resend the message to try and inform the receiver. Conversely, the history of received messages R act as the one-time-use resource, ensuring that we can only obtain each resource once. However, as described in Section 1.3, the escrow pattern, and the Aneris variant thereof, is not suited for transfer of dependent resources. In particular, while individual sockets protocols support the transfer of different resources through case analysis on the associated message, they do not inherently support dependencies between those resources.

The Actris ghost theory in an unreliable distributed settings. The Actris ghost theory lets us reason about reliable dependent resource transfer, however it does not immediately apply to unreliable distributed settings. The ghost theory tracks reliable buffers of messages in transit; and to apply the ghost theory, these reliable buffers have to be tied to physical objects. This was possible in prior work on Actris [Hinrichsen et al. 2022], as the ghost theory was applied to shared memory message-passing, built on top of lock-protected buffers. However, no such physical objects exist in a distributed setting! While sufficient ghost state can be employed to tie the logical buffers of the Actris ghost theory to the physical messages that are relayed over the network, it is not immediately clear how this can be achieved, and it does not solve the unreliable nature of the message exchanges. Additionally, as mentioned earlier, the Actris ghost theory imposes an obligation to strip multiple laters when applying its rules. In the prior work on Actris this was achieved by instrumenting the critical section of the implementation, guarded by a lock, with sufficient non-operative steps (skip instructions), that would each strip a later. This is not possible in a distributed setting, as the only way to share the session context is via atomically accessible invariants. Instead one needs to strip all of the laters imposed by the ghost theory during a single physical step. We describe how we deal with this challenge in Section 4.2.

3 RELIABLE COMMUNICATION LIBRARY API AND SPECIFICATION

In this section we present the API (Section 3.1) and the specification (Section 3.2) of the reliable communication library that we have implemented and verified, followed by the verification of the simple example presented in Figure 1 (Section 3.3).

3.1 Reliable Communication Library API

Figure 4 describes the API of the reliable communication library implementation. The API declares abstract data types of sockets and channel descriptors, and exposes the BSD socket-like primitives for client-server bidirectional (message-directed) communication.

```
type ('a, 'b) client_skt
type ('a, 'b) server_skt
type ('a, 'b) chan_descr
val mk_clt_skt : 'a serializer → 'b serializer → saddr → ('a, 'b) client_skt
val mk_srv_skt : 'a serializer → 'b serializer → saddr → ('a, 'b) server_skt
val listen : ('a, 'b) server_skt → unit
val accept : ('a, 'b) server_skt → ('a, 'b) chan_descr * saddr
val connect : ('a, 'b) client_skt → saddr → ('a, 'b) chan_descr
val send : ('a, 'b) chan_descr → 'a → unit
val try_recv : ('a, 'b) chan_descr → 'b option
val recv : ('a, 'b) chan_descr → 'b
```

Fig. 4. The API of the reliable communication library.

We make an explicit distinction between client_skt, the type of *active* sockets on which clients connect to a given server, server_skt, the type of *passive* sockets on which the servers listen for the incoming data from multiple clients, and chan_descr, the type of channel descriptors that clients and servers can use for reliable data transmission, once the clients' connection requests have been accepted by the server and the connection has been established.

The library is polymorphic in the types of values exchanged between the clients and server. This is achieved by making the library serialize the exchanged data internally, allowing the user directly to send and receive values of the chosen data types, instead of operating on strings, which is the standard type of message contents in Aneris. Thus the socket descriptor types are parameterized by a pair of types $('a,'b)$ and to create sockets, one must provide serializers for encoding/decoding strings to and from those data types.

The API of our library can be used following the usual workflow of reliable client-server communication: (1) by calling the listen function, the server is set to listen for incoming connection requests, which the server can accept, one at a time, by calling the accept function, which returns a new channel descriptor for each accepted connection; (2) each client connects to the server, by calling the connect function, which, when it terminates, returns a new channel descriptor on the client side; (3) once the connection is established, each side can use its own channel descriptor for reliable data transmission in both directions, by calling the `send` and `recv` functions.

3.2 Reliable Communication API and Specifications

Similar to how the OCaml API hides the implementation details of the RCLib, our specification, shown in Figure 5, hides the verification details that are irrelevant to the user. It does so by using a *dependent specification pattern*, in which the specifications of the API primitives are dependent on the *user parameters* (UP : RC_UserParams) provided by the user, and on the *abstract specification parameters* (S : RC_Resources UP) provided by the library itself.[3] To initialize the library, the user must supply the following four parameters:

- *srv*: the statically known socket address of the server;
- *prot*: the dependent separation protocol clients can use to interact with the server;
- *ss*: the serializer for the values sent by the server/received by clients;
- *cs*: the serializer for the values sent by clients/received by the server.

For brevity's sake, we simply write S.`srv` instead of UP.`srv`, whenever S : RC_Resources UP.

The initialization is captured formally by the RC-INIT-ALLOC rule. The rule is parametric in a freely picked instance of the user parameters UP, and yields an instance of the library provided abstract

[3] One can think of the dependent specification pattern as providing a logically specified module interface dependent on universally quantified user parameters, and existentially quantified abstract specification resources.

RC User Parameters and Resources:

$$\mathit{UP} \in \mathsf{RC\_UserParams} \triangleq \{\mathsf{srv} : \mathsf{Address};\quad \mathsf{prot} : \mathsf{iProto};\quad \mathsf{ss} : \mathsf{Serializer};\quad \mathsf{cs} : \mathsf{Serializer}\}$$

$$S \in \mathsf{RC\_Resources}\ (\mathit{UP} : \mathsf{RC\_UserParams}) \triangleq \left\{\begin{array}{rlrl} \mathsf{CanListen} : & \mathsf{Socket} \to \mathsf{iProp}; & \mathsf{SrvInit} : & \mathsf{iProp}; \\ \mathsf{Listens} : & \mathsf{Socket} \to \mathsf{iProp}; & \Phi_{\mathsf{srv}} : & \mathsf{Message} \to \mathsf{Prop} \\ \mathsf{CanConnect} : & \mathsf{Ip} \to \mathsf{Socket} \to \mathsf{iProp}; & & \end{array}\right\}$$

RC-INIT-ALLOC

$$\mathsf{True} \Rrightarrow \exists (S : \mathsf{RC\_Resources}\ \mathit{UP}).\ S.\mathsf{SrvInit}$$

Server Setup Specifications:

HT-MAKE-SERVER-SOCKET [S]

$$\left\{\begin{array}{l} S.\mathsf{srv} \Mapsto S.\Phi_{\mathsf{srv}} * S.\mathsf{SrvInit} * \\ \mathsf{FreeAddr}(S.\mathsf{srv}) * S.\mathsf{srv} \rightsquigarrow (\emptyset, \emptyset) \end{array}\right\}$$
$$\langle S.\mathsf{srv.ip};\ \mathsf{mk\_srv\_skt}\ S.\mathsf{ss}\ S.\mathsf{cs}\ S.\mathsf{srv}\rangle$$
$$\{w.\ \exists \mathit{skt}.\ w = \mathit{skt} * S.\mathsf{CanListen}\ \mathit{skt}\}$$

HT-LISTEN [S]

$$\{S.\mathsf{CanListen}\ \mathit{skt}\}$$
$$\langle S.\mathsf{srv.ip};\ \mathsf{listen}\ \mathit{skt}\rangle$$
$$\{S.\mathsf{Listens}\ \mathit{skt}\}$$

HT-ACCEPT [S]

$$\{S.\mathsf{Listens}\ \mathit{skt}\}\ \langle S.\mathsf{srv.ip};\ \mathsf{accept}\ \mathit{skt}\rangle\ \{w.\ \exists c, \mathit{sa}.\ w = (c, \mathit{sa}) * S.\mathsf{Listens}\ \mathit{skt}\ * c \xrightarrow[S.\mathsf{ss}]{S.\mathsf{srv.ip}} \overline{S.\mathsf{prot}}\}$$

Client Setup Specifications:

HT-MAKE-CLIENT-SOCKET [S]

$$\left\{\begin{array}{l} S.\mathsf{srv} \Mapsto S.\Phi_{\mathsf{srv}} * \mathsf{Unallocated}(\{\mathit{sa}\}) * \\ \mathsf{FreeAddr}(\mathit{sa}) * \mathit{sa} \rightsquigarrow (\emptyset, \emptyset) \end{array}\right\}$$
$$\langle \mathit{sa}.\mathsf{ip};\ \mathsf{mk\_clt\_skt}\ S.\mathsf{ss}\ S.\mathsf{cs}\ \mathit{sa}\rangle$$
$$\{w.\ \exists \mathit{skt}.\ w = \mathit{skt} * S.\mathsf{CanConnect}\ \mathit{sa}.\mathsf{ip}\ \mathit{skt}\}$$

HT-CONNECT [S]

$$\{S.\mathsf{CanConnect}\ \mathit{ip}\ \mathit{skt}\}$$
$$\langle \mathit{ip};\ \mathsf{connect}\ \mathit{skt}\ S.\mathsf{srv}\rangle$$
$$\{w.\ \exists c.\ w = c * c \xrightarrow[S.\mathsf{cs}]{\mathit{sa}.\mathsf{ip}} S.\mathsf{prot}\}$$

Reliable Data Transmission Specifications:

HT-RELIABLE-SEND

$$\left\{\begin{array}{l} c \xrightarrow[\mathit{ser}]{\mathit{ip}} \,!\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot} * \\ P[\vec{t}/\vec{x}] * \mathsf{Ser}\ \mathit{ser}\ (v[\vec{t}/\vec{x}]) \end{array}\right\}$$
$$\langle \mathit{ip};\ \mathsf{send}\ c\ (v[\vec{t}/\vec{x}])\rangle$$
$$\{c \xrightarrow[\mathit{ser}]{\mathit{ip}} \mathit{prot}[\vec{t}/\vec{x}]\}$$

HT-RELIABLE-TRY-RECV

$$\{c \xrightarrow[\mathit{ser}]{\mathit{ip}} \,?\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}\}$$
$$\langle \mathit{ip};\ \mathsf{try\_recv}\ c\rangle$$
$$\left\{\begin{array}{l} w.\ (w = \mathsf{None} * c \xrightarrow[\mathit{ser}]{\mathit{ip}} \,?\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) \lor \\ \quad (\exists \vec{y}.\ w = \mathsf{Some}\ (v[\vec{y}/\vec{x}]) * c \xrightarrow[\mathit{ser}]{\mathit{ip}} \mathit{prot}[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]) \end{array}\right\}$$

HT-RELIABLE-RECV

$$\{c \xrightarrow[\mathit{ser}]{\mathit{ip}} \,?\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}\}\ \langle \mathit{ip};\ \mathsf{recv}\ c\rangle\ \{w.\ \exists \vec{y}.\ w = v[\vec{y}/\vec{x}] * c \xrightarrow[\mathit{ser}]{\mathit{ip}} \mathit{prot}[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]\}$$

Fig. 5. The specifications of the Reliable Communication Library

specification parameters S, along with an initialisation resource $S.\mathsf{SrvInit}$. To use the verification framework a user is then expected to:

(1) Obtain the initial network resources via the Aneris adequacy theorem (Theorem 2.1);
(2) Initialise the reliable communication library via the RC-INIT-ALLOC and VS-CSQ rules;
(3) Allocate the static server socket interpretation with the library-provided server protocol $S.\mathsf{srv} \Mapsto S.\Phi_{\mathsf{srv}}$ using HT-SOCKET-INTERP-ALLOC;
(4) Distribute the server initialisation resource $S.\mathsf{SrvInit}$ to the server node, and the duplicable static server socket interpretation $S.\mathsf{srv} \Mapsto \overline{S.\mathsf{prot}}$ to all nodes.

Setup specifications. The specification of the server setup is given by the rules Ht-make-server-socket [S], Ht-listen [S], and Ht-accept [S]. The Ht-make-server-socket [S] rule takes the initialisation resource $S.\mathtt{SrvInit}$, the static server interpretation $S.\mathsf{srv} \Rrightarrow S.\Phi_{\mathsf{srv}}$, along with the primitive Aneris resources $\mathsf{FreeAddr}(S.\mathsf{srv})$ and $S.\mathsf{srv} \rightsquigarrow (\emptyset, \emptyset)$ to set up the server socket. As a result, we obtain the resource $S.\mathtt{CanListen}\ skt$ that can then be used to satisfy the precondition of the Ht-listen [S] rule. In return, the postcondition of the Ht-listen [S] rule yields the resource $S.\mathtt{Listens}\ skt$ which can then be passed to the precondition of the Ht-accept [S] rule in order to obtain the channel descriptor resource $c \xrightarrow[S.\mathsf{ss}]{S.\mathsf{srv.ip}} \overline{S.\mathtt{prot}}$ of the next incoming established connection, with the (dual of the) user picked protocol $S.\mathtt{prot}$. Note that the postcondition of the Ht-accept [S] rule both provides the user with the channel descriptor ownership and gives the $S.\mathtt{Listens}\ skt$ resource back (so that the accept function can be called again).

The specification of the client setup is given by the rules Ht-make-client-socket [S] and Ht-connect [S]. The former allows setting up the client socket, by supplying the static server socket interpretation $S.\mathsf{srv} \Rrightarrow S.\Phi_{\mathsf{srv}}$, and the primitive Aneris resources $\mathsf{FreeAddr}(sa)$, $sa \rightsquigarrow (\emptyset, \emptyset)$, and $\mathsf{Unallocated}(\{sa\})$, which yields the $S.\mathtt{CanConnect}\ sa.\mathsf{ip}\ skt$ resource. The latter then allows the client to connect to the server, consuming the $S.\mathtt{CanConnect}\ ip\ skt$ token to produce the channel endpoint ownership $c \xrightarrow[S.\mathsf{cs}]{sa.\mathsf{ip}} S.\mathtt{prot}$, with the initial protocol state $S.\mathtt{prot}$.

Reliable data transmission specifications. Once a session has been established between the server and client, they share the same specifications, based on the channel endpoint ownership fragment $c \xrightarrow[ser]{ip} prot$, where *prot* determines the current state of the session. Both sides can then exchange values in accordance with the protocol, using the Ht-reliable-send, Ht-reliable-try-recv, and Ht-reliable-recv rules. The rules are remniscent of the Actris ghost theory rules presented in Figure 3 (except that for send, we need to show that the value to be sent ($v[\vec{t}/\vec{x}]$) is serializable by the associated serializer *ser*).

3.3 A Simple Example: Verifying a String Length Server

To illustrate how the RCLib specifications can be used concretely, we consider the example presented in Figure 1, of a server that returns the length of each incoming string.

To prove that the assertion `assert (m1 = 5 && m2 = 4)` never fails, we prove two individual separation logic specifications for the server and client, compose them using a system node, and then apply the adequacy theorem (see section 2.2). The system node is defined as follows:

```
start (srv.ip) (server srv); start (clt.ip) (client clt srv)
```

The full formal specification and proof thereof can be found in our accompanying artifact [Gondelman et al. 2023]; we now give an overview of it. The crux of the verification is to use an appropriate dependent separation protocol, which in this example can be the `echo_prot` protocol presented in Section 1. We thus start by instantiating the RCLib with the following user parameters:

$$\mathsf{UP} \triangleq \{\mathtt{srv} := \mathtt{srv};\quad \mathtt{prot} := \mathtt{echo\_prot};\quad \mathtt{ss} := \mathtt{int\_ser};\quad \mathtt{cs} := \mathtt{str\_ser}\}$$

Here the $\mathsf{UP}.\mathtt{srv}$ is some globally known socket address, and the protocol (from the client's view) is `echo_prot`. The serialized values are strings (from client to server) and integers (from server to clients). The library then provides us with the resources $S : \mathsf{RC\_Resources}\ (\mathsf{UP})$ and the proof rules for RCLib primitives that we can use to verify the client and the server. We then show the following specifications for the client and server:

$$\left\{S.\mathsf{srv} \Rrightarrow S.\Phi_{\mathsf{srv}} * S.\mathtt{SrvInit} * \mathsf{FreeAddr}(S.\mathsf{srv}) * S.\mathsf{srv} \rightsquigarrow (\emptyset, \emptyset)\right\} \langle \mathsf{S.srv.ip};\ \mathtt{server}\ S.\mathtt{srv}\rangle \{\mathsf{False}\}$$
$$\left\{S.\mathsf{srv} \Rrightarrow S.\Phi_{\mathsf{srv}} * \mathsf{Unallocated}(\{sa\}) * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset)\right\} \langle sa.\mathtt{ip};\ \mathtt{client}\ sa\ \mathsf{S.srv}\rangle \{\mathsf{True}\}$$

Until the session has been established, both proofs are done by symbolic execution. Then, we can prove the server loops by Löb induction (a proof principle for reasoning about recursive definitions), by showing that at any given iteration, both loops end in the same state that they began. For the `accept_loop` this is straightforward, as the S.`Listens` *skt* token is preserved when applying Ht-accept [S]. For the `serve_loop` this is easy as well, as the `echo_prot` protocol recurses after two steps, so the proof boils down to showing that the body of the loop adheres to the `echo_prot` protocol. This is straightforward to show, using Ht-reliable-recv and Ht-reliable-send rules.

The verification of the client is a slightly more subtle, since the client sends two messages in a row, after which it awaits for two messages in a row, and as such this does not match syntactically with the `echo_prot`. However, it does so semantically, since the client's second send request and its first received response are independent, and so we can update the protocol[4] by using the subprotocol relation as we explained in Section 2.3. The return values dictated by the protocol (|"Carpe"| and |"Diem"|) then let us show that the assertions hold, which concludes the proof.

As an indication of the proof effort of verifications performed with the RCLib the program and proof of this example consists of $\sim$ 350 lines of Coq code.

4 IMPLEMENTING AND VERIFYING THE RELIABLE COMMUNICATION LIBRARY

In this section, we provide insight on how we implemented and verified the key parts of the RCLib w.r.t. the specifications given in Figure 5. We focus on how we achieve the unreliable dependent resource transfer specified by the dependent protocols via a novel proof pattern—the *session escrow pattern*—which conceptually merges the distributed sharing of spatial resources via the escrow pattern with the reliable dependent resource transfer of the dependent separation protocols. We first give an overview of how we implemented the reliable communication library (Section 4.1). We then cover the session escrow pattern, and how it resolves key limitations of Aneris and Actris when applied to reliable distributed transfer (Section 4.2). Finally, we give an overview of how we tie the session escrow pattern to the physical code to verify the send and receive operations (Section 4.3).

4.1 Implementation of the RCLib Channel Descriptor

The RCLib is implemented directly on top of Aneris's primitive unreliable socket handlers. It employs an asynchronous server-client architecture, where the server serves multiple clients *on the same socket handle*. Once a connection is established the asymmetric nature is hidden via symmetric *channel descriptors*, on which the client and server operate identically. The implementation consists of three distinct parts, describe in detail below:

- The connection step: Initiating the session via unreliable socket primitives.
- The channel descriptors: Symmetric interface for the session endpoints.
- Internal network procedures: Tagging and tracking sequence ids, retransmission, etc.

Implementation of the connection step. The server and client sockets are initialised with their respective socket operation mk_srv_skt S.ss S.cs S.srv and mk_clt_skt S.ss S.cs *sa*, which bundle the pre-determined serializers (S.ss and S.cs) together with a new socket, which is allocated and bound via the unreliable socket primitives of AnerisLang. The server is set to listen using the listen *skt* operation. This operation allocates an *accept* buffer for new available sessions, and starts a loop that awaits incoming server messages on the server socket. Server messages can either be connection requests, from new clients, or session messages and acknowledgements from existing clients. Connection requests are sent by clients using connect *skt* S.srv to the statically known server address S.srv. When a connection request is received, the server initialises a new session by allocating a channel descriptor for the channel (described momentarily), enqueueing it into the *accept* buffer, and responding (with retransmission) that the connection was successful. Once

[4]A single Coq tactic resolves the subprotocol relation, updates the protocol and executes the second send request.

the connection acknowledgement has been received, the connect skt S.srv function starts a loop that awaits incoming session messages on the client socket. The function additionally allocates and returns a new channel descriptor. Finally, the accept skt function is used on the server-side to dequeue and return the first channel descriptor in the queue.

Implementation of the channel descriptors. We represent a channel descriptor as a 4-tuple consisting of $(\ell_{sbuf}, \ell_{rbuf}, slk, rlk)$. The reference ℓ_{sbuf} stores a send buffer $sbuf$, implemented as a queue, which stores the values to be sent over the network. Each call to `send` c v then simply enqueus v to $sbuf$. The reference ℓ_{rbuf} stores a receive buffer $rbuf$, implemented as a queue, containing values coming from the other session endpoint, in the order that they were originally sent (by virtue of the underlying network implementation). Each user's call to `recv` c then simply loops until a value is available, dequeues it from the queue $rbuf$, and returns it to the user. Finally, the slk, rlk are locks guarding the send and receive buffer respectively (since those buffers are shared between the internal procedures and user's calls to the send/receive operations). Using buffers allows us to implement send and receive in a way that is simple, network agnostic, and identical for the client and server (and it also simplifies verification).

Implementation of the internal network procedures. The internal network procedures of the server and client concurrently forward values from/to the buffers of the channel descriptors.

In parallel to user calls to send, the internal sending procedure (a non-terminating loop) keeps (re)transmitting the contents of $sbuf$ over the network via the unreliable network primitives of AnerisLang. To achieve sequential ordering the messages are ascribed a *sequence id* which reflects the order in which they were originally enqueued into the send buffer and which lets the other endpoint accept them in the order they were sent. To this end, the internal procedure maintain a reference ℓ_{sid} to a *sequence id lower bound sid* that reflects the sequence id of the first message in $sbuf$ (initially 0). The messages $v_0 \dots v_i \dots v_{(|sbuf|-1)}$ of $sbuf$ are thus indexed by $sid + i$. To avoid retransmitting messages forever, the internal procedure accepts acknowledgement messages. When an acknowledgement message is received, the $sbuf$ queue is pruned and the sid is updated accordingly. If the acknowledgement message is less than sid the message is discarded. Otherwise, all messages with a sequence id less than the acknowledgement are removed from $sbuf$ and sid is updated to the acknowledgement id, being the new lowest sequence id. Outbound messages are serialized using the outbound serializer stored in the server/client sockets respectively.

The internal receiving procedure (again a non-terminating loop) awaits incoming messages, and deserializes them using the inbound serializer stored in the server/client sockets respectively. Inbound messages are filtered based on their sequence id, to ensure messages are only received once. To this end, the loop uses a reference ℓ_{aid} which stores the current *acknowledgement id aid*, which is the index that the next incoming message is expected to have. If the index of an inbound message matches aid, the message payload is enqueued into the receive queue $rbuf$, and an acknowledgement message with aid is sent back. If the received index is lower than aid, an acknowledgement message with the current aid is still sent back to notify the sender that they can prune their buffer and thus stop retransmitting the deprecated message. If the received index is higher than the current sequence id, the message is simply discarded.

Finally, the server handles incoming requests by cross referencing the address of the incoming session request with the list of channel descriptors, and processes the incoming message with respect to the corresponding sequence id and receive queue.

4.2 Unreliable Dependent Resource Transfer with the Session Escrow Pattern

To verify the unreliable dependent resource transfer in an unreliable network we merge the ideas behind the escrow pattern, already used in Aneris, with the dependent separation protocols, already

SESCROW-INIT
$$\Rrightarrow \exists \chi.\, \texttt{ses\_own}\ \chi\ \texttt{left}\ 0\ 0\ \mathit{prot} * \texttt{ses\_own}\ \chi\ \texttt{right}\ 0\ 0\ \overline{\mathit{prot}}$$

SESCROW-SEND
$$\frac{\texttt{ses\_own}\ \chi\ s\ n\ m\ (\mathbf{!}\,(\vec{x}:\vec{\tau})\ \langle v\rangle\{P\}.\,\mathit{prot}) * P[\vec{t}/\vec{x}]}{\Rrightarrow \texttt{ses\_own}\ \chi\ s\ (n+1)\ m\ (\mathit{prot}[\vec{t}/\vec{x}]) * \texttt{ses\_idx}\ \chi\ s\ n\ (v[\vec{t}/\vec{x}])}$$

SESCROW-RECV
$$\frac{\texttt{ses\_own}\ \chi\ s\ n\ m\ (\mathbf{?}(\vec{x}:\vec{\tau})\ \langle v\rangle\{P\}.\,\mathit{prot}) * \texttt{ses\_idx}\ \chi\ \bar{s}\ m\ w}{\Rrightarrow \exists(\vec{y}:\vec{\tau}).\, \texttt{ses\_own}\ \chi\ s\ n\ (m+1)\ (\mathit{prot}[\vec{y}/\vec{x}]) * w = v[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]}$$

SESCROW-DUP
$$\frac{\texttt{ses\_idx}\ \chi\ s\ n\ v}{\texttt{ses\_idx}\ \chi\ s\ n\ v * \texttt{ses\_idx}\ \chi\ s\ n\ v}$$

HT-STEP-MODALITY
$$\frac{\{P\}\ \langle \mathit{ip};\ e\rangle\ \{w.\, Q\}}{\{P * \Rrightarrow R\}\ \langle \mathit{ip};\ e\rangle\ \{w.\, Q * R\}}$$

Fig. 6. The session escrow pattern and step modality (Mask details omitted[5]).

used in Actris. The result of this is the novel so-called "Session Escrow Pattern", which has been formalised partially via the Actris ghost theory. The pattern leverages the unreliable spatial resource transfer of the escrow pattern, namely that we can asynchronously commit and release resources, and track the state of the transfer via a duplicable witness, which we can replay over the network. The pattern also leverages the reliable dependent resource transfer of the dependent separation protocols, by allowing the ascription of expressive protocols of dependent sessions.

The intuition behind the pattern is that it, much like the Actris ghost theory, lets us initialise a session described by a dependent separation protocol, which acts as an agreement between the separate channel descriptors about what messages and associated resources will be sent in either direction, and in which order. A transfer made by either side is evidenced by a duplicable witness, much alike the escrow pattern, which can be sent indefinitely over the network, until it is received by the other side. Once received, the witness can then be used by the other side to obtain the transferred resources. The pattern is formally presented in Figure 6. The pattern operates on two types of resources, $\texttt{ses\_own}\ \chi\ s\ n\ m\ \mathit{prot}$ and $\texttt{ses\_idx}\ \chi\ s\ n\ v$. The χ argument is an identifier that associates the resources with each other. The s argument signifies which side of the session the resource belongs to. The first resource has three additional arguments: n, m, and prot. The n and m arguments capture the number of messages that have been sent and received by the endpoint, to track how far the endpoint is in the protocol. The prot argument capture the local view of the protocol. The second resource has two additional arguments: n and v. The n argument captures the index of the message, while the v argument captures the value that the message is associated with.

The proof rules of the pattern are quite similar to the Actris ghost theory. The SESCROW-INIT rule initialises a new session, yielding a ghost resource for both endpoints, $\texttt{ses\_own}\ \chi\ \texttt{left}\ 0\ 0\ \mathit{prot}$ and $\texttt{ses\_own}\ \chi\ \texttt{right}\ 0\ 0\ \overline{\mathit{prot}}$, which have initially sent and received zero messages. The protocol prot is chosen freely for one endpoint, while the other gets the dual $\overline{\mathit{prot}}$, similar to the Actris ghost theory. The SESCROW-SEND rule takes an endpoint resource with a sending protocol ($\texttt{ses\_own}\ \chi\ s\ n\ m\ (\mathbf{!}\,\vec{x}:\vec{\tau}\ \langle v\rangle\{P\}.\,\mathit{prot})$), and the resources described by the protocol ($P[\vec{t}/\vec{x}]$), for some instantiation of its binders ($\vec{t}:\vec{\tau}$). It returns the endpoint resource with the updated protocol and sending index ($\texttt{ses\_own}\ \chi\ s\ (n+1)\ m\ (\mathit{prot}[\vec{t}/\vec{x}])$), along with a witness that the message has been sent, tracking the corresponding message index and value ($\texttt{ses\_idx}\ \chi\ s\ n\ (v[\vec{t}/\vec{x}])$). The SESCROW-RECV rule takes

[5]The actual session escrow pattern is constructed using an Iris invariant and is specified with a "namespace" $\mathcal{N}$. Iris uses namespaces to avoid opening the same invariant twice. We refer the interested reader to [Jung et al. 2016].

an endpoint resource with a receiving protocol ($\texttt{ses\_own}\ \chi\ s\ n\ m\ (?\vec{x} : \vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot})$), and a witness from the other endpoint that corresponds to the current receive index ($\texttt{ses\_idx}\ \chi\ \bar{s}\ m\ w$). It returns the endpoint resource with the updated protocol and receive index ($\texttt{ses\_own}\ \chi\ s\ n\ (m + 1)\ (\mathit{prot}[\vec{t}/\vec{x}])$), and the resources described by the protocol ($P[\vec{y}/\vec{x}]$), for some instantiation of the identifiers ($\vec{y} : \vec{\tau}$). The sescrow-dup rule captures that the witnesses can be freely duplicated.

Finally, the rules are defined using a novel step modality $\models\!\!\leadsto R$, as opposed to the multiple laters in the Actris ghost theory. Intuitively, $\models\!\!\leadsto R$ holds if we can obtain R after taking a step of the operational semantics. This is made precise by the associated rule Ht-step-modality, which states that one can resolve R in the postcondition of a Hoare triple, when having $\models\!\!\leadsto R$ in the precondition.

While the step modality may seem similar to the later modality, it leverages recent discoveries that allow resolving multiple later modalities during one step of the operational semantics [Matsushita et al. 2022; Mével et al. 2019; Spies et al. 2022]. In particular, the modality abstracts over the concrete number of laters that needs to be resolved, by internalising that we can resolve *enough* laters, at every operational step. This abstraction over the number of laters is imperative for defining the session escrow pattern. It would be impossible to state the above ghost theory rules, as either endpoint is unable to determine the number of laters they would need to resolve, since that number is related to the number of inbound messages, which cannot be inferred from the local state.

4.3 Verifying the Reliable Communication Library

With the session escrow pattern presented above, we now give an overview of how we can verify the reliable communication library. Similar to the implementation, the verification can be considered in three parts; the connection step, the channel descriptors, and the internal network procedures.

Verifying the connection step. While established sessions have disjoint resources, the resources are initially allocated together (using the sescrow-init rule). This happens on the server side, during the handshake, when the server transfers the client's resource (the $\texttt{ses\_own}\ \chi\ \texttt{left}\ 0\ 0\ \mathit{prot}$ resource) to the client, using the Aneris rules. The client and server must agree on the session protocol before the handshake (to satisfy the Aneris rules, which require mutual agreement on the socket interpretations), which holds since the (statically known) server only serves a single pre-determined protocol *prot*. This kind of distributed channel creation is in contrast with the message-passing concurrency instantiation of Actris ghost theory, where both channel endpoints are stored on the same node, and can thus be created logically and physically at the same time.

Verifying the channel descriptors. To verify the channel descriptors, we associate the physical counters *sid* / *aid* and buffer sizes $|\mathit{sbuf}|$ / $|\mathit{rbuf}|$ with the counters of the session escrow context as follows $\texttt{ses\_own}\ \chi\ s\ (\mathit{sid} + |\mathit{sbuf}|)\ (\mathit{aid} - |\mathit{rbuf}|)\ \mathit{prot}$. We enforce this correspondence using the Iris ghost theory for exclusive agreement, consisting of two resources $\boxed{\bullet\, n}^{\gamma}$ and $\boxed{\circ\, m}^{\gamma}$, which always agree $\boxed{\bullet\, n}^{\gamma} * \boxed{\circ\, m}^{\gamma} \Rightarrow n = m$, and can be updated together $\boxed{\bullet\, n}^{\gamma} * \boxed{\circ\, m}^{\gamma} \Rrightarrow \boxed{\bullet\, o}^{\gamma} * \boxed{\circ\, o}^{\gamma}$.

We additionally associate each element $v_0 \dots v_i \dots v_{(|\mathit{sbuf}|-1)}$ of the send buffer with an outbound session escrow witness $\texttt{ses\_idx}\ \chi\ s\ (\mathit{sid} + i)\ v_i$, and each element $w_0 \dots w_j \dots w_{(|\mathit{rbuf}|-1)}$ of the receive buffer with an inbound session escrow witness $\texttt{ses\_idx}\ \chi\ \bar{s}\ (\mathit{aid} - |\mathit{rbuf}| + j)\ w_j$. The significance of the indices are elaborated upon at the end of this section.

To formally verify the channel descriptors we use an existing lock library of Aneris, which enforce *lock invariants* that hold between any access to the critical section of the related lock. With the above correspondences in place, we define the lock invariants of the buffers as follows:

$$\begin{array}{l} \texttt{sbuf\_inv}\ \chi\ \gamma_n\ \mathit{ip}\ \ell_{\mathit{sbuf}}\ \ell_{\mathit{sid}}\ \mathit{ser}\ s \triangleq \exists \mathit{sbuf}, \mathit{sid}. \\ \quad \ell_{\mathit{sbuf}} \overset{\mathit{ip}}{\mapsto}_{\mathsf{q}} \mathit{sbuf} * \ell_{\mathit{sid}} \overset{\mathit{ip}}{\mapsto} \mathit{sid} * \boxed{\circ\, (\mathit{sid} + |\mathit{sbuf}|)}^{\gamma_n} * \\ \quad \mathop{\Large *}_{i \mapsto v \in \mathit{sbuf}} \texttt{Ser}\ \mathit{ser}\ v * \texttt{ses\_idx}\ \chi\ s\ (\mathit{sid} + i)\ v \end{array} \qquad \begin{array}{l} \texttt{rbuf\_inv}\ \chi\ \gamma_m\ \mathit{ip}\ \ell_{\mathit{rbuf}}\ \ell_{\mathit{aid}}\ s \triangleq \exists \mathit{rbuf}, \mathit{aid}. \\ \quad \ell_{\mathit{rbuf}} \overset{\mathit{ip}}{\mapsto}_{\mathsf{q}} \mathit{rbuf} * \ell_{\mathit{aid}} \overset{\mathit{ip}}{\mapsto} \mathit{aid} * \boxed{\circ\, (\mathit{aid} - |\mathit{rbuf}|)}^{\gamma_m} * \\ \quad \mathop{\Large *}_{j \mapsto w \in \mathit{rbuf}} \texttt{ses\_idx}\ \chi\ \bar{s}\ (\mathit{aid} - |\mathit{rbuf}| + j)\ w \end{array}$$

The lock invariants reflect the physical state of their respective buffer (using a "queue resource" connective $\ell \overset{ip}{\mapsto}_{\mathsf{q}} q$) and counter (using the points-to-predicate $\ell \overset{ip}{\mapsto} v$). They additionally govern one part of the exclusive agreement ghost state $\boxed{\circ\ n}^{\gamma}$. Finally, they capture the correspondence between elements and session escrow witnesses. Notably, the iterated separation conjunction $\mathop{\ast}_{i \mapsto v \in \vec{v}}$ asserts ownership of separate resources at each element v of the buffer $\vec{v}$ at index i. With the lock invariants in place we can define the channel descriptor resource as follows:

$$\begin{aligned} c \xrightarrow[ser]{ip} prot \triangleq \exists \chi, \gamma_n, \gamma_m, s, n, m, \ell_{sid}, \ell_{aid}.&\\ &c = (\ell_{sbuf}, \ell_{rbuf}, slk, rlk) \ast \texttt{ses\_own}\ \chi\ s\ n\ m\ prot \ast \boxed{\bullet\ n}^{\gamma_n} \ast \boxed{\bullet\ m}^{\gamma_m} \ast\\ &\texttt{is\_lock}\ ip\ slk\ (\texttt{sbuf\_inv}\ \chi\ \gamma_n\ ip\ \ell_{sbuf}\ \ell_{sid}\ ser\ s) \ast\\ &\texttt{is\_lock}\ ip\ rlk\ (\texttt{rbuf\_inv}\ \chi\ \gamma_m\ ip\ \ell_{rbuf}\ \ell_{aid}\ s) \end{aligned}$$

The descriptor resource captures the session escrow context $\texttt{ses\_own}\ \chi\ s\ n\ m$ along with the other part of the exclusive agreement ghost states $\boxed{\bullet\ n}^{\gamma_n}$ and $\boxed{\bullet\ m}^{\gamma_m}$. Moreover, the descriptor resource also keeps a copy of the duplicable lock predicates of both buffers.

With the definition of the channel descriptor resource we can verify the send and receive functions of the reliable communication library. Whenever we send, we first obtain the internals of the send buffer lock invariant (when acquiring the lock), and unify the counter n with $sid + |sbuf|$. We then simply use the session escrow send rule SESCROW-SEND to obtain a new witness $\texttt{ses\_idx}\ \chi\ s\ (sid + |sbuf|)\ v$ at the next sequence id, which we delegate to the send buffer when enqueueing the value. Conversely, whenever we receive, we unify m with $aid - |rbuf|$, dequeue the first element w_0 of the receive buffer, and derive that it has the witness $\texttt{ses\_idx}\ \chi\ \bar{s}\ (aid - |rbuf|)\ w$. We can then use the witness along with the session escrow receive rule SESCROW-RECV to obtain the resources specified by the protocol.

Verifying the internal network procedures. Verifying the internal layers of the server and clients primarily involve relaying the session escrow witnesses using the primitive unreliable network rules of Aneris. Ultimately, this means that we have to pick an appropriate socket interpretation. Sidestepping the connection and acknowledgement messages, the socket interpretations asserts that messages (1) are serializable using the pre-determined serializers, (2) are pairs of sequence identifiers i and payload values v, and (3) are associated with a corresponding witness:

$$\begin{aligned} \Phi_{\texttt{clt}} \triangleq \lambda m.&\\ &(\exists i, v. S.\texttt{ss}\ m.\texttt{str}\ (\texttt{inl}(i, v)) \ast\\ &\quad \texttt{ses\_idx}\ \chi\ \texttt{right}\ i\ v) \lor\\ &\langle acknowledgement\_message \rangle \end{aligned} \qquad \begin{aligned} \Phi_{\texttt{srv}} \triangleq \lambda m.&\\ &(\exists i, v. S.\texttt{cs}\ m.\texttt{str}\ (\texttt{inl}(i, v)) \ast\\ &\quad \texttt{ses\_idx}\ \chi\ \texttt{left}\ i\ v) \lor\\ &\langle acknowledgement\_message \rangle \lor\\ &\langle connection\_message \rangle \end{aligned}$$

With this, the verification follows from using Aneris's primitive rules for sending and receiving along with preserving the invariants of the send and receive buffers. Notably, we are able to guarantee that we only enqueue fresh values into the receive buffers, by filtering inbound messages via the acknowledgement id *aid*.

As mentioned in Section 4.1, the implementation uses concurrency internally; verifying this concurrency was achieved using existing Iris / Aneris verification techniques and thus we do not further detail the verification thereof.

5 REMOTE PROCEDURE CALL LIBRARY

To demonstrate the expressivity of the RCLib specs (Section 3), we now consider the specification and verification of a multi-threaded remote procedure call (RPC) library. In Section 6 we will then show how this library itself is used to facilitate the formal development of clients and applications that make use of it.

RPC API:

```
type ('a, 'b) rpc
val rpc_start : 'b serializer → 'a serializer → saddr → ('a → 'b) → unit
val rpc_connect : 'a serializer → 'b serializer → saddr → saddr → ('a, 'b) rpc
val rpc_make_request : ('a, 'b) rpc → 'a → 'b
```

RPC User Parameters and Resources:

$$UP \in \texttt{RPC\_UserParams} \triangleq \left\{\begin{array}{lll} \texttt{srv} : \mathsf{Address}; & \texttt{ReqData} : \mathsf{Type}; & \texttt{RepData} : \mathsf{Type}; \\ \texttt{qs} : \mathsf{Serializer}; & \multicolumn{2}{l}{\texttt{pre} : \mathsf{Val} \to \texttt{ReqData} \to \mathsf{iProp};} \\ \texttt{rs} : \mathsf{Serializer}; & \multicolumn{2}{l}{\texttt{post} : \mathsf{Val} \to \texttt{ReqData} \to \texttt{RepData} \to \mathsf{iProp}} \end{array}\right\}$$

$$S \in \texttt{RPC\_Resources}\ (UP : \texttt{RPC\_UserParams}) \triangleq \{\texttt{SrvInit} : \mathsf{iProp};\quad \texttt{CanConnect} : \mathsf{Address} \to \mathsf{iProp};\quad \Phi_{\mathsf{srv}} : \mathsf{Message} \to \mathsf{Prop}\}$$

RPC-INIT-ALLOC

$$\mathsf{True} \Rrightarrow \exists (S : \texttt{RPC\_UserParams}\ UP).\ S.\texttt{SrvInit}$$

RPC Specifications:

HT-RPC-START [S]

$$\left\{\begin{array}{l} S.\texttt{srv} \Mapsto S.\Phi_{\mathsf{srv}} * S.\texttt{SrvInit} * \\ \mathsf{FreeAddr}(S.\texttt{srv}) * S.\texttt{srv} \rightsquigarrow (\emptyset, \emptyset) * \\ \texttt{rpc\_process\_spec}\ S\ \mathit{proc} \end{array}\right\}$$
$$\langle S.\texttt{srv.ip};\ \texttt{rpc\_start}\ S.\texttt{rs}\ S.\texttt{qs}\ S.\texttt{srv}\ \mathit{proc}\rangle$$
$$\{\mathsf{True}\}$$

$$\texttt{rpc\_process\_spec}\ S\ \mathit{proc} \triangleq \forall qv, qd.$$
$$\{S.\texttt{pre}\ qv\ qd\}$$
$$\langle S.\texttt{srv.ip};\ \mathit{proc}\ qv\rangle$$
$$\{rv.\ \exists rd.\ \mathsf{Ser}\ S.\texttt{rs}\ rv * S.\texttt{post}\ rv\ qd\ rd\}$$

HT-RPC-CONNECT [S]

$$\left\{\begin{array}{l} \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \\ S.\texttt{srv} \Mapsto S.\Phi_{\mathsf{srv}} * \mathsf{Unallocated}(\{sa\}) \end{array}\right\}$$
$$\langle sa.\texttt{ip};\ \texttt{rpc\_connect}\ S.\texttt{qs}\ S.\texttt{rs}\ sa\ S.\texttt{srv}\rangle$$
$$\{rpc.\ S.\texttt{CanRequest}\ sa.\texttt{ip}\ rpc\}$$

HT-RPC-REQUEST [S]

$$\left\{\begin{array}{l} S.\texttt{CanRequest}\ ip\ rpc * \\ S.\texttt{pre}\ qv\ qd * \mathsf{Ser}\ S.\texttt{qs}\ qv \end{array}\right\}$$
$$\langle ip;\ \texttt{rpc\_make\_request}\ rpc\ qv\rangle$$
$$\{rv.\ S.\texttt{CanRequest}\ ip\ rpc * \exists rd.\ S.\texttt{post}\ rv\ qd\ rd\}$$

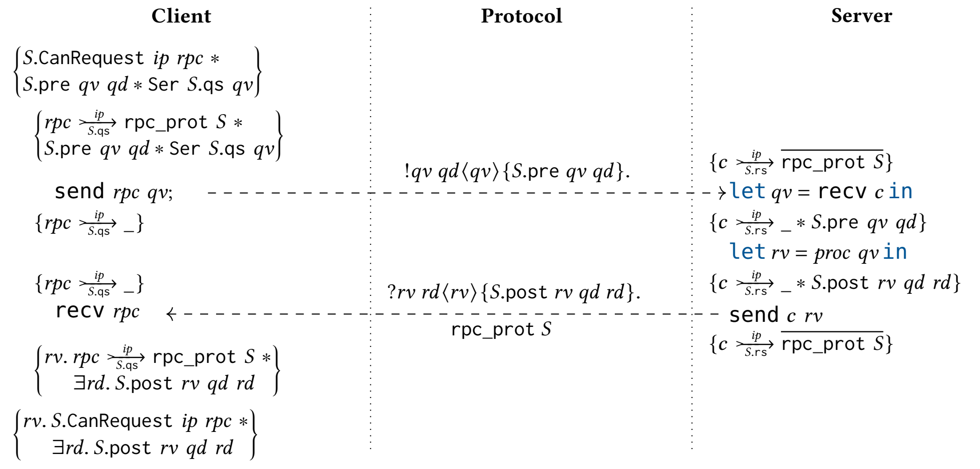
Fig. 7. Specifications for the RPC library.

We have implemented, specified and verified a variant of such an RPC service. Our variant exposes just one service handler, but it allows the types of the client's request and the server's response to be *polymorphic*. In particular, when instantiating those types with sum-types $\tau_q^1 + \tau_q^2$ for requests (and $\tau_r^1 + \tau_r^2$ for responses), we can effectively encode an RPC service that handles multiple procedure calls, *e.g.*, as a pair of procedures of type $\tau_q^1 \to \tau_r^1$ and $\tau_q^2 \to \tau_r^2$.

Figure 7 shows the API and the specifications of our RPC library. The RPC library can be initialised by calling `rpc_start`, which is parametric in the serializers for the request- and response data types, the socket address of the server, and the implementation of the procedure that will be used to handle the incoming requests. To call the procedure remotely, the clients must first connect to the server, by calling `rpc_connect`, which yields the RPC handle *rpc*. The handle is then used as an argument of `rpc_make_request` along with some input data to make a request.

5.1 Specifications of the RPC library

The specifications of the RPC are parametric in the user provided parameters (UP : `RPC_UserParams`), which most importantly consist of the universally established server address (S.`srv`), and the logical data types of the requests and replies (S.`ReqData` and S.`RepData`). Additionally, the user must


Client
Protocol
Server
$\{S.\mathsf{CanRequest}\ ip\ rpc * S.\mathsf{pre}\ qv\ qd * \mathsf{Ser}\ S.\mathsf{qs}\ qv\}$
$\{rpc \xrightarrow[S.\mathsf{qs}]{ip} \mathsf{rpc\_prot}\ S * S.\mathsf{pre}\ qv\ qd * \mathsf{Ser}\ S.\mathsf{qs}\ qv\}$
send $rpc\ qv$;
$\{rpc \xrightarrow[S.\mathsf{qs}]{ip} \_\}$
$\{rpc \xrightarrow[S.\mathsf{qs}]{ip} \_\}$
recv rpc
$\{rv.\ rpc \xrightarrow[S.\mathsf{qs}]{ip} \mathsf{rpc\_prot}\ S * \exists rd.\ S.\mathsf{post}\ rv\ qd\ rd\}$
$\{rv.\ S.\mathsf{CanRequest}\ ip\ rpc * \exists rd.\ S.\mathsf{post}\ rv\ qd\ rd\}$
$!qv\ qd\langle qv\rangle\{S.\mathsf{pre}\ qv\ qd\}.$
$?rv\ rd\langle rv\rangle\{S.\mathsf{post}\ rv\ qd\ rd\}.$
rpc_prot S
$\{c \xrightarrow[S.\mathsf{rs}]{ip} \overline{\mathsf{rpc\_prot}\ S}\}$
let qv = recv c in
$\{c \xrightarrow[S.\mathsf{rs}]{ip} \_ * S.\mathsf{pre}\ qv\ qd\}$
let $rv = proc\ qv$ in
$\{c \xrightarrow[S.\mathsf{rs}]{ip} \_ * S.\mathsf{post}\ rv\ qd\ rd\}$
send $c\ rv$
$\{c \xrightarrow[S.\mathsf{rs}]{ip} \overline{\mathsf{rpc\_prot}\ S}\}$
Fig. 8. The reliable communication of the RPC library

determine the serializers to be used for the request and reply values (S.qs and S.rs), so that the client and server can serialize and deserialize the exchanged messages without coordination. Finally, the user must provide pre- and post-condition predicates (S.pre and S.post) that relate the request and reply values with their corresponding logical values.

In return the RPC library provides the abstract specification parameters (S : RPC_Resources UP), which consist of SrvInit, $S.\Phi_{\mathsf{srv}}$, and the S.CanRequest $ip\ rpc$ resources. The rpc library is initialised using the RPC-INIT-ALLOC rule, similarly to the RCLib approach.

To start the RPC service using rpc_start the user must use the HT-RPC-START [S] specification, which needs the static server socket interpretation $S.\mathsf{srv} \Mapsto S.\Phi_{\mathsf{srv}}$, the SrvInit resource, along with the primitive Aneris resources FreeAddr(S.srv) and $S.\mathsf{srv} \rightsquigarrow (\emptyset, \emptyset)$. Additionally, the user must prove that the procedure $proc$ satisfies the specification defined by rpc_process_spec. Indeed, this specification ensures that the procedure function handles the incoming requests correctly. In particular, rpc_process_spec states that the procedure argument qv must satisfy the provided precondition S.pre $qv\ qd$, and that the results rv must satisfy the provided postcondition S.post $rv\ qd\ rd$. In other words, when starting the server, the user must prove rpc_process_spec for the procedure function that they choose.

To connect to the RPC service using the rpc_connect operation, clients must use the HT-RPC-CONNECT [S] rule, to give up the server socket interpretation $S.\mathsf{srv} \Mapsto S.\Phi_{\mathsf{srv}}$, along with the primitive Aneris resources FreeAddr(S.srv), $S.\mathsf{srv} \rightsquigarrow (\emptyset, \emptyset)$, and Unallocated($\{sa\}$). The specification then yields the S.CanRequest $ip\ rpc$ resource for the returned RPC handle rpc. Finally, the HT-RPC-REQUEST [S] specification captures how the client can make requests when in possession of the S.CanRequest $ip\ rpc$ resource. Additionally, the argument qv must satisfy the provided precondition S.pre $qv\ qd$, and qv must be serializable by the provided request serializer S.qs. In return the client obtains the resources of the postcondition S.post $rv\ qd\ rd$ for the returned value rv.

5.2 Verification of the RPC library

The main challenge of verifying the RPC library is to show that the specification of the client's rpc_make_request function follows from the user provided proof of the request handler at the server side, cf. rpc_process_spec. We address this challenge by using a dependent separation protocol which specifies the delegation of the handler call to the server:

$$\begin{aligned}\texttt{rpc\_prot}\ (S : \texttt{RPC\_Resources}\ \mathit{UP})\ \triangleq\ & \\ \mu rec.\ !\ (qv : \mathsf{Val})(qd : S.\texttt{ReqData})\ \langle qv\rangle\{S.\texttt{pre}\ qv\ qd\}. & \\ ?(rv : \mathsf{Val})(rd : S.\texttt{RepData})\ \langle rv\rangle\{S.\texttt{post}\ rv\ qd\ rd\}.\ rec & \end{aligned}$$

The protocol describes (from the clients point of view) the request-reply communication. The client first sends a value *qv*, which is related to the request data *qd* by the provided $S.\texttt{pre}\ qv\ qd$ predicate. The server will then reply with a value *rv*, related to some reply data *rd* and the original request data *qd* by the provided $S.\texttt{post}\ rv\ qd\ rd$ predicate.

Figure 8 sketches the proof of how this protocol connects the specifications of the client's local and remote calls to verify Ht-rpc-request [S]. First, the abstract resource $S.\texttt{CanRequest}\ ip\ rpc$ is unfolded, to obtain the channel endpoint ownership $rpc \xrightarrow[S.\mathsf{qs}]{ip} (\texttt{rpc\_prot}\ S)$. Then the resources for the request value ($S.\texttt{pre}\ qv\ qd$) are transferred along the request. On the server side, when the resources are received, they are supplied to the procedure *proc*, yielding the reply value *rv* and the resources $S.\texttt{post}\ rv\ qd\ rd$, which are then sent back to the client (in accordance with the protocol). On the client side, the processed request and resources are finally received and returned. As the protocol completed one cycle of recursion and returns to the initial state, it is packed back into the abstract resource $S.\texttt{CanRequest}\ ip\ rpc$, so that the postcondition of the `rpc_make_request` holds. In summary, the dependent separation protocols of RCLib make it quite simple to verify the implementation of the RPC library!

6 LAZY REPLICATION WITH LEADER-FOLLOWERS

To illustrate the power of our approach to reason about reliable network components in a highly modular way, we now show how to specify and verify an implementation of the leader-followers key-value store KVS, which we build directly on top of the RPC library. As we will see, our modular approach enables us to verify KVS without having to reason about the UDP network (handled by the RCLib) or the RCLib protocols and specifications (handled by the RPC library)!

Concretely, the leader-followers KVS we present is a replicated KVS that provides different guarantees for read and write operations. The entire system consists of a central server node, called the *leader* and the multiple replica nodes, called *followers* – the idea is that a client has to direct all write requests to the leader while they have a choice to direct read operations at the leader or any of the followers. Importantly, to ensure consistency of reads from followers (w.r.t writes directed to the leader), the latter and all the followers, are guaranteed to agree upon, and preserve, the order of write operations. This is achieved by having leader to register all the write operations as a part of its state. This state is then *lazily* replicated by the followers servers which periodically poll the state of the leader and store a local copy of it. Note that because replication is lazy, the system is more available, but provides weaker consistency guarantees for the reads from followers. Indeed, while the read operation directed at the leader is guaranteed to always return the most up-to-date value, a read directed at a follower may return a stale value.

6.1 Implementation of the Leader-Followers KVS

Leader and followers are implemented directly on top of the RPC library. Thus we only need to implement handlers which, upon clients' requests, write (at the leader) or read (at the leader or follower) the local state of the server (here we use instantiate our RPC library with sum-types so as encode a service that handles multiple procedure calls).

The local state of each node consists of a key-value table together with a log of all write events observed by that server. The idea is that the primary state of the KVS is the log. The key-value table is a memoization table to optimize read operations which simply look up the value in the table instead of seeking the latest written value to the requested key in the log. Hence, the write operation on the leader, in addition to adding the write event to the log, also updates the local table. Similarly, when a follower receives a new write event from the leader, in addition to adding it to its local log, it updates its local copy of the table.

The interaction between the leader and the followers is also implemented using the RPC library where the leader assumes the role of the server for followers which periodically make a request to the leader asking for the next available log entry they have not seen yet. The programs for both the leader and followers are concurrent programs, *e.g.*, the leader runs two different threads, one for serving clients and another one for serving followers. These programs use locks to protect the data structures shared between different threads running on each server.

6.2 Specification of the Leader-Followers KVS

We first consider a simple version of the system with only one server: the leader. In this setting, we can give simple specifications to read and write, similar to those for local heap-allocated references:

LEADER-ONLY-WRITE-SPEC
$$\left\{k \mapsto^{\mathsf{ldr}} vo\right\} \langle ip;\ write\ k\ v\rangle \left\{k \mapsto^{\mathsf{ldr}} \mathsf{Some}\ v\right\}$$

LEADER-ONLY-READ-SPEC
$$\left\{k \mapsto^{\mathsf{ldr}}_{q} vo\right\} \langle ip;\ read\ k\rangle \left\{x.\ k \mapsto^{\mathsf{ldr}}_{q} vo * x = vo\right\}$$

Here the $k \mapsto^{\mathsf{ldr}} vo$ proposition, where vo is an optional value, asserts ownership over the key k in the KVS and indicates its value (None indicates that no writes have taken place on that particular key). The proposition $k \mapsto^{\mathsf{ldr}}_{q} vo$ is the fractional variant where ownership is only asserted for a fraction $0 < q \in \mathbb{Q} \leq 1$.

The specs given above for reading and writing in fact remain sound for interacting with the leader even in the presence of followers. The values read from followers can correspond to old write operations which have since been overwritten. In order to express this intuition formally we introduce propositions in our logic for tracking the history of all write operations in the form of a sequence of *write events*. A write event, we, is a tuple consisting of the target key in the KVS, the written value, as well as its logical time, *i.e.*, its index in the history of write events observed by the system. We write $we.key$ and $we.value$ for the key and value of the write event respectively. Furthermore, we write $h{\downarrow_k}$ for the optional value of the last (latest) write event in history h whose key is k. We use the observation proposition $Obs(\mathsf{DB}, h)$, defined in terms of Iris resources, to indicate that the history h has been observed at the server whose address is DB; this server could either be the leader or a follower. The important intuition here is that write operations are immediately observed on the leader while they are only observed on followers if they have occurred before the point in time when said follower has last polled and copied the state of the leader. Observation propositions only express the knowledge that a certain history has been observed and are thus persistent in the technical Iris sense, which implies that they are duplicable: $Obs(\mathsf{DB}, h) \dashv\vdash Obs(\mathsf{DB}, h) * Obs(\mathsf{DB}, h)$. In addition to introducing observations we also let points-to predicates specify the optional write event corresponding to the key instead of an optional value. That is, in the proposition $k \mapsto^{\mathsf{kvs}} wo$ (our form of points-to proposition for the system featuring followers), wo is an optional write event, which allows us to express stronger guarantees for the write operation.

Following an approach similar to Gondelman et al. [2021], we use Iris invariants to express the relationship between the logical state of each key on the leader, exposed to the client as $k \mapsto^{\mathsf{kvs}} wo$, the logical state of what is observed by each server, exposed to the client as $Obs(\mathsf{DB}, h)$, and the physical state (stored in the memory) of each server, not exposed to the client. The following tables give a summary of the building blocks used in the specification of leader and followers:

WRITE-SPEC

$$\{k \mapsto^{\mathsf{kvs}} wo * Obs(\mathsf{DB}_{ld}, h) * h{\downarrow_k} = wo\}\ \langle ip;\ write\ k\ v\rangle\ \left\{\begin{array}{r}\exists hf, we.\ we.key = k * we.value = v * hf{\downarrow_k} = \mathsf{None} * \\ Obs(\mathsf{DB}_{ld}, h \mathbin{+\!\!+} hf \mathbin{+\!\!+} [we]) * k \mapsto^{\mathsf{kvs}} \mathsf{Some}\ we\end{array}\right\}$$

LEADER-READ-SPEC

$$\{k \mapsto^{\mathsf{kvs}}_q wo\}\ \langle ip;\ read\ k\rangle\ \{x.\ k \mapsto^{\mathsf{kvs}}_q wo * ((x = \mathsf{None} \land wo = \mathsf{None}) \lor (\exists we.\ x = \mathsf{Some}\ we.value \land wo = \mathsf{Some}\ we))\}$$

FOLLOWER-READ-SPEC

$$\{Obs(\mathsf{DB}_{fl}, h)\}\ read_{fl}\ k\ \left\{\begin{array}{l}x.\ \exists h'.\ h \le_p h' * Obs(\mathsf{DB}_{fl}, h') * \\ \quad \left((x = \mathsf{None} \land h'{\downarrow_k} = \mathsf{None}) \lor (\exists we.\ x = \mathsf{Some}\ we.value \land h'{\downarrow_k} = \mathsf{Some}\ we)\right)\end{array}\right\}$$

Fig. 9. The specification for the write operation and the read operation for both the leader and followers.

Proposition	**Intuitive meaning**
$k \mapsto^{\mathsf{kvs}} wo$	Asserts exclusive ownership over the key k with the last write event being wo. Note that wo is an optional value and can be None which indicates that no value has ever been written to k.
$Obs(\mathsf{DB}, h)$	This persistent proposition asserts the knowledge that history h has been observed by the server whose address is DB.
$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}}$	Relates the resources underlying $k \mapsto^{\mathsf{kvs}} v$ and $Obs(\mathsf{DB}, h)$ and enables tying these to physical states through local invariants (one invariant per server) which are not exposed to the client.

Symbol	**Meaning**
we	Ranges over write events.
wo	Ranges over optional write events, *i.e.*, it is either None or Some *we*.
$write$	The write function.
$read$	The read function for leader.
$read_{fl}$	The read function for follower *fl*.
DB	Ranges over server addresses: leader or follower.
DB_{ld}	The addresses of the leader.
DB_{fl}	The address of follower *fl*.

These are the important properties of observations (the full list can be found in the accompanying Coq formalization):

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}} * k \mapsto^{\mathsf{kvs}}_q wo \Rrightarrow k \mapsto^{\mathsf{kvs}}_q wo * \exists h.\ Obs(\mathsf{DB}_{ld}, h) * h{\downarrow_k} = wo \qquad \text{(observe-at-leader)}$$

$$\boxed{\mathsf{GlobalInv}}^{\mathcal{N}} * Obs(\mathsf{DB}, h) \Rrightarrow \exists h'.\ Obs(\mathsf{DB}_{ld}, h') * h \le_p h' \qquad \text{(leader-observes-first)}$$

$$Obs(\mathsf{DB}, h) * Obs(\mathsf{DB}', h') \vdash h \le_p h' \lor h' \le_p h \qquad \text{(linear-order)}$$

The property (observe-at-leader) states that the current value stored by the leader is always observed by the leader; the history where this write event is the last write event with key k is observed on the leader. Note how this property is stated using the update modality, $\Rrightarrow$, which allows for accessing invariants to obtain the necessary information since points-to propositions, observations, and the physical states of servers are all tied together using such invariants. The property (linear-order) captures that all servers, the leader and the followers, agree on the order of observed write events; as such one history is always a prefix of the other.

The specifications for writing to the KVS, reading from the leader, and reading from the followers are given in Figure 9. Note how the specification for reading a key on the leader, LEADER-READ-SPEC, is exactly the same as the leader-only situation, LEADER-ONLY-READ-SPEC. On the other hand, the write spec, WRITE-SPEC, is strengthened compared to LEADER-ONLY-WRITE-SPEC. It states that having $k \mapsto^{\mathsf{kvs}} wo$, the write event added as the result of this call, is the first write event after wo.

The specification for reading from a follower, FOLLOWER-READ-SPEC, states that after reading we get the knowledge that the observed history on that follower is possibly extended in a way such that the returned optional write event is consistent with this observed history — the extended history is the one observed at the moment the read operation was carried out on the follower.

Note how the specifications for the read and write operations, despite the implementation of the KVS being based on that of the RPC library and in turn on the reliable communication library

```
let do_writes () =
  write "x" 37; write "y" 1

let rec wait_on_read k v =
  let res = read_fl k in
  if res = Some v
  then ()
  else wait_on_read k v
```

```
let do_reads () =
  wait_on_read "y" 1;
  let vx = read_fl "x" in
  assert (vx = Some 37)

let client0 () = do_writes ()
let client1 () = do_reads ()
client0 () ||| client1 ()
```

Fig. 10. Example Client of Leader-Followers.

and ultimately Aneris's network primitives, do not mention any of these dependencies or their specs. This demonstrates that our modular verification approach enables proper encapsulation of modules (what Krogh-Jespersen et al. [2020] refer to as vertical modularity). Note also that the leader-only specifications can be derived from the general specs (see Appendix A).

Client Example. Figure 10 shows an example of the program using the KVS. It consists of two clients running in parallel on two different nodes (written with three parallel vertical lines). We assume that the leader and the followers have been initialized prior to running these clients. One client, `client0`, performs two write operations, 37 to x followed by 1 to y. The other client, `client1`, perform two read operations directed at a follower. It first waits until it observes the value 1 on y and then asserts that x has value 37. Note that the program order in `do_writes` implies that the second write *causally depends on* the first write. See the accompanying Coq formalization for a formal proof of this example client; the proof guarantees that the assert in `do_reads` will not fail. The example above demonstrates that reading from a follower satisfies *monotonic reads*, *monotonic writes*, and *writes follow reads* guarantees [Terry et al. 1994], but does not provide the *read-your-writes* guarantee, as the leader does not synchronize with followers during the writes.

6.3 Verification of the Leader-Followers KVS

The crux of the verification is to *(a)* give concrete definitions of the abstract predicates, *e.g.*, $Obs(\mathsf{DB}, h)$ and $k \mapsto^{\mathsf{kvs}} wo$, *(b)* instantiate the specifications of the RPC library for handlers, and *(c)* show the Hoare triples for the handlers as ascribed by the RPC library. We omit a description of those steps (see Appendix A and accompanying Coq formalization) and just mention one elided nuance, namely that the specifications we have presented for the read and write operations do not capture the fact that these operations are logically atomic. To verify the example above, one needs to use logical atomicity (in order to be able to open invariants around read and write operations). And indeed, in our Coq formalization, our specifications for read and write do capture the logical atomicity; technically the specifications are given in the so-called HOCAP-style [Svendsen et al. 2013], from which the read and write specifications presented in Figure 9 can easily be derived.

7 RELATED WORK

Reliable Transport Protocols in Verification of Distributed Systems. In recent years, there have been several verification frameworks to reason about implementations and/or high-level models of distributed systems. Some of these works focus on high-level properties of distributed applications *assuming* that the underlying transport layer of the verification framework is reliable, *e.g.*, [Koh et al. 2019; Sergey et al. 2018; Zhang et al. 2021] and the first version of Aneris framework [Gondelman et al. 2021; Krogh-Jespersen et al. 2020]. Other works that focus on high-level properties of distributed applications [Hawblitzel et al. 2017; Nieto et al. 2022; Wilcox et al. 2015] also treat the reliable communication as a part of the verification process to some extent.

Nieto et al. [2022] implement a reliable causal broadcast (RCB) library in Aneris and use it to implement op-based conflict-free replicated data types (CRDTs). Their implementation uses vector clocks to achieve causal reliable delivery. While RCB can be used for reliable communication, it is not suitable for client-server communication. Firstly, since the RCB implements broadcast, in order to isolate clients from one another, the server would need to create a new instance of RCB for each client, and somehow match the standard API for reliable communication (listen, accept, ...) as we do. Secondly, and more importantly, the RCB specification does not expose the dependencies in the session communication; instead it only tracks the causality relation between the delivered messages and the already received messages as sets, not sequences. In contrast, our RCLib specification is session-type based and tracks dependencies using Actris's dependent separation protocols. The Verdi framework [Wilcox et al. 2015] proposes a methodology to verify distributed systems that relies on a notion of *verified transformers*. One such transformer is a Sequence Numbering Transformer that allows ensuring that messages are delivered at most once, similar to the guarantees provided by our RCLib. However, in Verdi verified transformers are stated in a high-level domain-specific language which abstracts over implementation details such as node-local concurrency or message serialization, and the reasoning is done in terms of traces on the high-level semantics. In contrast, developing RCLib in Aneris enables both the modular verification of a realistic implementation of a reliable transport communication layer (horizontal modularity) and the modular verification of the clients of the RCLib (vertical modularity). Moreover, some of the existing verification systems assume that the shim connecting the analysis framework to executable code is reliable [Lesani et al. 2016; Wilcox et al. 2015]. That can limit guarantees about the verified code and lead to the discrepancies between the high-level specification, verification tool, and shim of such verified distributed systems [Fonseca et al. 2017].

Verification of Reliable Transport Layer Protocols. There has been several works focusing on showing correctness of protocols for reliable communication. Smith [1996]'s work is one of the earliest on formal verification of communication protocols. Bishop et al. [2006] provide HOL specification and symbolic-evaluation testing for TCP implementations. Compton [2005] presents Stenning's protocol verified in Isabelle. Badban et al. [2005] presents verification of a sliding window protocol in μCRL. None of those works however capture the reliability guarantees in a logic in a modular way that facilitates reasoning about clients of those protocols. In contrast, our work both verifies the reliable transport layer as a library and provides a modular high-level specification for reasoning about distributed libraries and applications that require reliable communication. This is illustrated in our work by the case studies such as verification of the RPC library and leader-followers KVS, for which our work, to the best of our knowledge, is the first formal modular verification of such distributed applications. Broadly speaking, considering the existing verified prior work, we believe that it can conceptually be ported to using the RCLib, with minor modifications regarding how the reliable transfer is encoded and specified. However, carrying out this port in a mechanized setting is in practice non-trivial, and heavily relies on the frameworks in question. Porting Iris-based proofs would likely be easier, as a lot of the mechanization shares the same foundation, but reusing proofs from a framework like Verdi would require more effort.

Session Types in Distributed Systems. Session types, since their inception by Honda [1993], have primarily been concerned with idealised reliable communication, where messages are never dropped, duplicated, or received out of order. Castro-Perez et al. [2019] developed a toolchain for "transport-independent" multi-party session typed endpoints in Go. They show how their theory applies to channel endpoints that may communicate locally (via shared memory) and in a distributed setting (via TCP). Miu et al. [2021] developed a toolchain for generating TypeScript WebSocket code for session type-checked TCP-based reliable communication in a distributed setting. Their system guarantees communication safety and deadlock freedom, for which they provide a paper proof. Recent work considers variations of unreliable communication, focused on constructing new session type variants for handling the setting in question. Kouzapas et al. [2019] develops a session type variant for such an unreliable setting where messages can be lost (although they are never duplicated or arrive out of order). Their system handles message loss by tagging messages with a sequence id where, when a failure is detected, the session catches up to the protocol through some parametric failure handling mechanism. They provide such a mechanism, where a default value of the expected type is returned, after which the sequence id is increased. In contrast to related work, our work establishes a high-level reliable communication library built on top of a low-level unreliable network, which is then given reliable specifications, via conventional session-type like protocols. We are unaware of existing work that takes this approach.

8 CONLUSION AND FUTURE WORK

In this paper we have demonstrated the maturity of the Aneris distributed separation logic and the genericity of the Actris dependent separation protocol framework, by combining them to implement and verify a suite of reliable network components on top of low-level unreliable semantics. Each component specification is encapsulated as an abstraction; no details about their building blocks are exposed, even when these consist of other libraries. While we deem our low-level unreliable semantics to be a step towards verification of more realistic languages, we find that the RCLib implementation could be further improved from future extensions. The implementation of the reliable communication library includes a mechanism for retransmitting messages until an acknowledgement is received. This is crucial, as messages could otherwise be lost in the network, never to be retransmitted, resulting in any blocking receive halting indefinitely. The Aneris logic however does not give us any formal guarantees about progress, and so cannot verify that our implementation of retransmission actually ensures progress. It would thus be interesting to investigate whether one can obtain any such progress guarantees for the library by using the Trillium refinement logic [Timany et al. 2021]. Trillium allows for proving refinements between the executions of the program and a user-defined model, and has been used to prove *eventual consistency* for a Conflict-Free Replicated Data Type (CRDT) in conjunction with Aneris.

The RCLib assumes that established connections are never closed, neither graciously, nor because of an abrupt connection loss, *e.g.* due to a remote's crash. Lifting those assumptions would allow obtaining an even more realistic implementation, *e.g.* with the possibility of closing the channel endpoints and connection reestablishment. For the latter, it would also be interesting to consider how our specifications could be adapted to consider the possibility of crashes, *e.g.* by integrating a crash-sensitive logic such as Perennial [Chajed et al. 2019]) into our framework. The implementation is not partition-tolerant, as any partitioning between the server and one of its client would prevent further communication between them. It would be interesting to investigate methods for achieving fault-tolerance in Aneris, *e.g.* by having a cluster of nodes acting as the server, so the clients can *broadcast* to the entire cluster, rather than communicating with a singular node. This would effectively handle partitions, as other nodes in the cluster could relay the message to the server, and help in the development of fault-tolerant libraries (*e.g.*, multi-consensus). Finally, our system does not consider network security. It would be interesting to investigate the verification of secure reliable channels, where the connection is provably secure after the initial handshake.

ACKNOWLEDGMENTS

We are grateful to Chet Murthy for helpful discussions. This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

A LAZY REPLICATION WITH LEADER-FOLLOWERS

A.1 Deriving the Leader-Only Spec

The leader-only specifications, LEADER-ONLY-WRITE-SPEC and LEADER-ONLY-READ-SPEC, can be derived from the general specs, WRITE-SPEC and LEADER-READ-SPEC, by defining the leader-only version of the points-to proposition as follows:

$$k \mapsto^{\mathsf{ldr}} vo \triangleq \begin{cases} k \mapsto^{\mathsf{kvs}} \mathsf{None} & \text{if } vo = \mathsf{None} \\ \exists wo.\ k \mapsto^{\mathsf{kvs}} \mathsf{Some}\ we * we.value = v & \text{if } vo = \mathsf{Some}\ v \text{ for some value } v \end{cases}$$

Note how the leader-only read function returns the value of the write event returned by the read function. The LEADER-ONLY-READ-SPEC spec follows straightforwardly from LEADER-READ-SPEC. To see how LEADER-ONLY-WRITE-SPEC follows from WRITE-SPEC note how we can use (observe-at-leader) to obtain that there exists a history h such that $h{\downarrow_k} = wo$ whenever we have $k \mapsto^{\mathsf{kvs}} wo$, which we get by unfolding the definition of $k \mapsto^{\mathsf{ldr}} vo$, and a case analysis on whether vo is None or Some v.

A.2 Concrete definitions of the abstract predicates used in leader-followers

We start by defining two sets of propositions in terms of Iris resources using Iris's so-called authoritative resource algebra and fractional resource algebras. These resource constructions are standard and hence we will not get into the details of these constructions; see Jung et al. [2018] for similar constructions, *e.g.*, the resource construction for relating the contents of the physical heap to separation logic's standard points-to propositions. Iris's authoritative resource algebra allows us to construct resources that can be split into two parts, a so-called full part and a so-called fragment part. The idea is that the fragments must always be *included* in the full part — the notion of included depends on the precise construction of the resource as we will explain below. These two sets of propositions are as follows:

Proposition	Intuition
$KWT\ (M)$	Tracks global view of the mapping from keys to their latest write events maintained by the leader.
$k \mapsto^{\mathsf{kvs}} wo$	As before; the write event always agrees with, *i.e.*, is included in, M in $KWT\ (M)$.
$Log_{\mathsf{G}}(\mathsf{DB}, h)$	Tracks the writes observed on server DB in the global invariant. Agrees with Log_{S} and Log_{L}.
$Log_{\mathsf{S}}(\mathsf{DB}, h)$	Tracks the writes observed on server DB in the local invariant of the server. Agrees with Log_{G} and Log_{L}.
$Log_{\mathsf{L}}(\mathsf{DB}, h)$	Tracks the writes observed on server DB in the proof of correctness of RPC handlers. Agrees with Log_{G} and Log_{S}.
$Obs(\mathsf{DB}, h)$	As before; the history h is a prefix of, *i.e.*, is included in, the history tracked in Log_{G}, Log_{S}, and Log_{L}.

Here the propositions $KWT\ (M)$ and $k \mapsto^{\mathsf{kvs}} wo$ are defined as an instance of the authoritative resource algebra where the former is defined the full part and the latter defined as a fragment. Similarly, the propositions $Log_{\mathsf{G}}(\mathsf{DB}, h)$, $Log_{\mathsf{S}}(\mathsf{DB}, h)$, and $Log_{\mathsf{L}}(\mathsf{DB}, h)$ are defined as the full part of an instance of the authoritative resource algebra (split into three different parts) while the proposition $Obs(\mathsf{DB}, h)$ is defined as a fragment in the same resource algebra.

The rules governing these propositions are shown in Figure 11. The rules capture how the inclusions of the underlying authoritative resource algebras are reflected for the propositions (notably in rules TABLE-LOOKUP, LOGS-AGREE, and OBS-PREFIX), and how they are preserved when resources are updated (notably in rules TABLE-UPDATE and OBS-UPDATE).

Given these propositions we can define the global and local invariants as follows:[6]

$$\mathsf{GlobalInv} \triangleq \exists M, h.\ KWT\ (M) * Log_{\mathsf{G}}(\mathsf{DB}_{ld}, h) * LogMapConsistent(h, M) *$$
$$\mathop{\huge *}_{fl \in Fs} \exists h'.\ Log_{\mathsf{G}}(\mathsf{DB}_{fl}, h') * h' \leq_p h$$

[6]The local invariant is essentially stated as a lock invariant. See [Birkedal and Bizjak 2017] for locks in Iris.

$$\textsc{Table-lookup}\quad KWT\,(M) * k \mapsto^{\mathsf{kvs}} wo \vdash M(k) = wo \qquad \textsc{Table-update}\quad KWT\,(M) * k \mapsto^{\mathsf{kvs}} wo \Rrightarrow KWT\,(M[k \mapsto wo']) * k \mapsto^{\mathsf{kvs}} wo'$$

$$\textsc{Logs-agree}\ \frac{X, Y \in \{\mathrm{G}, \mathrm{S}, \mathrm{L}\} \qquad X \neq Y}{Log_X(\mathrm{DB}, h) * Log_Y(\mathrm{DB}, h') \vdash h = h'} \qquad \textsc{Obs-prefix}\ \frac{X \in \{\mathrm{G}, \mathrm{S}, \mathrm{L}\}}{Log_X(\mathrm{DB}, h) * Obs(\mathrm{DB}, h') \vdash h' \leq_p h}$$

$$\textsc{Obs-update}\ \frac{h \leq_p h'}{Log_{\mathrm{G}}(\mathrm{DB}, h) * Log_{\mathrm{S}}(\mathrm{DB}, h) * Log_{\mathrm{L}}(\mathrm{DB}, h) \Rrightarrow Log_{\mathrm{G}}(\mathrm{DB}, h') * Log_{\mathrm{S}}(\mathrm{DB}, h') * Log_{\mathrm{L}}(\mathrm{DB}, h') * Obs(\mathrm{DB}, h')}$$

Fig. 11. Rules governing the internal leader-followers library propositions.

$$\begin{aligned}\mathsf{LocalInv}_{\mathrm{DB}} \triangleq \exists M, h, v, v'.\ & Log_{\mathrm{S}}(\mathrm{DB}, h) * LogMapConsistent(h, M) * \\ & \ell_{tbl_{\mathrm{DB}}} \overset{\mathrm{DB}}{\longmapsto} v * isMap(v, M) * \ell_{log_{\mathrm{DB}}} \overset{\mathrm{DB}}{\longmapsto} v' * isSeq(v', h)\end{aligned}$$

The global invariant states that there is a map M that is our global view of the state of the leader. It is consistent with the history observed by the leader. Also, the history observed by each follower is a prefix of the history of the leader. The local invariant on the other hand states that there is a map that is consistent with the history observed by the server and that this map is physically stored, as the value v, in the memory location $\ell_{tbl_{\mathrm{DB}}}$. Similarly, it asserts that the server physically stores the sequence that is the history h, as the value v', in the memory location $\ell_{log_{\mathrm{DB}}}$.

REFERENCES

Bahareh Badban, Wan J. Fokkink, Jan Friso Groote, Jun Pang, and Jaco van de Pol. 2005. Verification of a sliding window protocol in μCRL and PVS. *Formal Aspects Comput.* 17, 3 (2005), 342–388. https://doi.org/10.1007/s00165-005-0070-0

Lars Birkedal and Aleš Bizjak. 2017. Lecture Notes on Iris: Higher-Order Concurrent Separation Log. http://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf. (2017).

Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. 2006. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 55–66. https://doi.org/10.1145/1111037.1111043

David Castro-Perez, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed programming using role-parametric session types in go: statically-typed endpoint APIs for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.* 3, POPL (2019), 29:1–29:30. https://doi.org/10.1145/3290342

Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 243–258. https://doi.org/10.1145/3341301.3359632

Michael Compton. 2005. Stenning's Protocol Implemented in UDP and Verified in Isabelle. In *Theory of Computing 2005, Eleventh CATS 2005, Computing: The Australasian Theory Symposium, Newcastle, NSW, Australia, January/February 2005 (CRPIT, Vol. 41)*, Mike D. Atkinson and Frank K. H. A. Dehne (Eds.). Australian Computer Society, 21–30. http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV41Compton.html

Alan Fekete, Nancy Lynch, Yishay Mansour, and John Spinelli. 1993. The Impossibility of Implementing Reliable Communication in the Face of Crashes. *J. ACM* 40, 5 (nov 1993), 1087–1107. https://doi.org/10.1145/174147.169676

Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/3064176.3064183

Léon Gondelman, Simon Oddershede Gregersen, Abel Nieto, Amin Timany, and Lars Birkedal. 2021. Distributed causal memory: modular specification and verification in higher-order distributed separation logic. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434323

Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, and Lars Birkedal. 2023. Verifying Reliable Network Components in a Distributed Separation Logic with Dependent Separation Protocols (Archived Artifact).

https://doi.org/10.5281/zenodo.8121688.

James N Gray. 1979. A discussion of distributed systems. (1979).

Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. 2013. Failure Recovery: When the Cure Is Worse Than the Disease. In *14th Workshop on Hot Topics in Operating Systems, HotOS XIV, Santa Ana Pueblo, New Mexico, USA, May 13-15, 2013*. https://www.usenix.org/conference/hotos13/session/guo

J Y Halpern. 1987. Using Reasoning About Knowledge to Analyze Distributed Systems. *Annual Review of Computer Science* 2, 1 (1987), 37–68. https://doi.org/10.1146/annurev.cs.02.060187.000345

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2017. IronFleet: Proving Safety and Liveness of Practical Distributed Systems. *Commun. ACM* 60, 7 (June 2017), 83–92. https://doi.org/10.1145/3068608

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 6:1–6:30. https://doi.org/10.1145/3371074

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2022. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. *Log. Methods Comput. Sci.* 18, 2 (2022). https://doi.org/10.46298/lmcs-18(2:16)2022

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 715)*, Eike Best (Ed.). Springer, 509–523. https://doi.org/10.1007/3-540-57208-2_35

Naghmeh Ivaki, Nuno Laranjeiro, and Filipe Araujo. 2018. A Survey on Reliable Distributed Communication. *Journal of Systems and Software* 137 (03 2018), 713–. https://doi.org/10.1016/j.jss.2017.03.028

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 256–269. https://doi.org/10.1145/2951913.2951943

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

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. 17:1–17:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.17

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2019. From C to interaction trees: specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, Assia Mahboubi and Magnus O. Myreen (Eds.). ACM, 234–248. https://doi.org/10.1145/3293880.3294106

Dimitrios Kouzapas, Ramunas Gutkovas, A. Laura Voinea, and Simon J. Gay. 2019. A Session Type System for Asynchronous Unreliable Broadcast Communication. *CoRR* abs/1902.01353 (2019). arXiv:1902.01353 http://arxiv.org/abs/1902.01353

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

Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 357–370. https://doi.org/10.1145/2837614.2837622

Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 841–856. https://doi.org/10.1145/3519939.3523704

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

Anson Miu, Francisco Ferreira, Nobuko Yoshida, and Fangyi Zhou. 2021. Communication-safe web programming in TypeScript with routed multiparty session types. In *CC '21: 30th ACM SIGPLAN International Conference on Compiler Construction, Virtual Event, Republic of Korea, March 2-3, 2021*, Aaron Smith, Delphine Demange, and Rajiv Gupta (Eds.). ACM, 94–106. https://doi.org/10.1145/3446804.3446854

Abel Nieto, Léon Gondelman, Alban Reynaud, and Lars Birkedal. 2022. Modular Verification of Op-Based CRDTs in Separation Logic. *Proc. ACM Program. Lang.* OOPSLA (2022). Accepted for publication..

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.* 2, POPL (2018), 28:1–28:30. https://doi.org/10.1145/3158116

M. A. S. Smith. 1996. Formal Verification of Communication Protocols. In *FORTE*.

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Modular Reasoning about Separation of Concurrent Data Structures. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 169–188. https://doi.org/10.1007/978-3-642-37036-6_11

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94), Austin, Texas, USA, September 28-30, 1994.* 140–149. https://doi.org/10.1109/PDIS.1994.331722

Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Léon Gondelman, Abel Nieto, and Lars Birkedal. 2021. Trillium: Unifying Refinement and Higher-Order Distributed Separation Logic. *CoRR* abs/2109.07863 (2021). arXiv:2109.07863 https://arxiv.org/abs/2109.07863

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 357–368. https://doi.org/10.1145/2737924.2737958

Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference) (LIPIcs, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2021.32

Received 2023-03-01; accepted 2023-06-27