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, 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 OCaml on top of UDP and features characteristic traits of existing communication protocols, such as a simple handshaking protocol, bidirectional channels, and retransmission/acknowledgement mechanisms. We specify the library in the Aneris distributed separation logic using a distributed variant of 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 *sequentially consistent lazily replicated key-value store with leader-followers* and some clients thereof. Our development is highly modular – each component is verified relative to specifications of the components it uses (not the implementation). All the results we present are formalized in the Coq proof assistant.

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. 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]

Different transport layer libraries share two common traits: (1) they all 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 mechanized verification in proof assistants.

[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.

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, Portugal, mjp.pereira@fct.unl.pt; Amin Timany, Aarhus University, Denmark, timany@cs.au.dk; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

However, most of this research is situated on one of two ends of a spectrum of how the reliable communication (when it is required) is treated.

On one end, existing work focuses on high-level properties of distributed applications *assuming* that the underlying transport layer of the verification framework is reliable, *e.g.*, [Gondelman et al. 2021; Krogh-Jespersen et al. 2020; Sergey et al. 2018], or assuming 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].

On the other end of the spectrum, existing work focuses on showing 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]) without capturing 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, by connecting distributed applications to an unreliable network via a high-level modular specification of a verified implementation of a reliable network communication library, verified on top of an unreliable network.* Concretely, we use Aneris [Krogh-Jespersen et al. 2020], a distributed higher-order separation logic, to present the first modular specification and foundational verification of an OCaml implementation of a transport-layer-level reliable communication library. Our implementation uses UDP for unreliable network communication and the verification of the implementation leverages Aneris' facilities for reasoning about UDP-like unreliable communication.[2]

A key point of using a reliable transport layer library is to simplify programming of applications on top of it. Hence, it should also be expected that our specifications of the reliable communication library can similarly simplify *reasoning* about applications built on top of the library, by providing more abstract and simpler reasoning patterns than the low-level Aneris reasoning patterns. We achieve this by formulating our specifications of the reliable communication library in terms of a distributed variant of the so-called *dependent separation protocols*, which we integrate with Aneris via the Actris framework [Hinrichsen et al. 2022] from which the protocols originate.

To demonstrate the application and expressivity of our specifications, we implement and verify several non-trivial distributed applications on top of our reliable communication library. In the remainder of this introduction we give a more detailed overview of the technical development in the paper and summarize our contributions.

1.1 Overview of the Technical Development and Contributions

Figure 1 gives a graphical overview of the work presented in this paper. As shown in the left side of the figure, the reliable communication library and the clients thereof are implemented in a subset of OCaml, on top of an extensible library of simple data structures and message serialization, and a simple fixed shim that primarily defines OCaml wrappers around the UDP network primitives and concurrency primitives such as locks and monitors.

The reliable communication library (RCLib) supports asynchronous asymmetric channel creation (using a 4-way handshake *à la* SCTP) and is implemented using standard techniques such as sequence identifiers, retransmissions/acknowledgments, and channel descriptors for bidirectional data transmission. On top of RCLib we implement an RPC service, and a sequentially consistent lazily replicated key-value store with leader-followers. Finally, we have implemented and verified some example client programs of the leader-followers key-value store.

[2]The first publication on Aneris Krogh-Jespersen et al. [2020] assumed duplicate protection of the network; that assumption has since been lifted, making the Aneris network model very close to UDP unreliable communication.

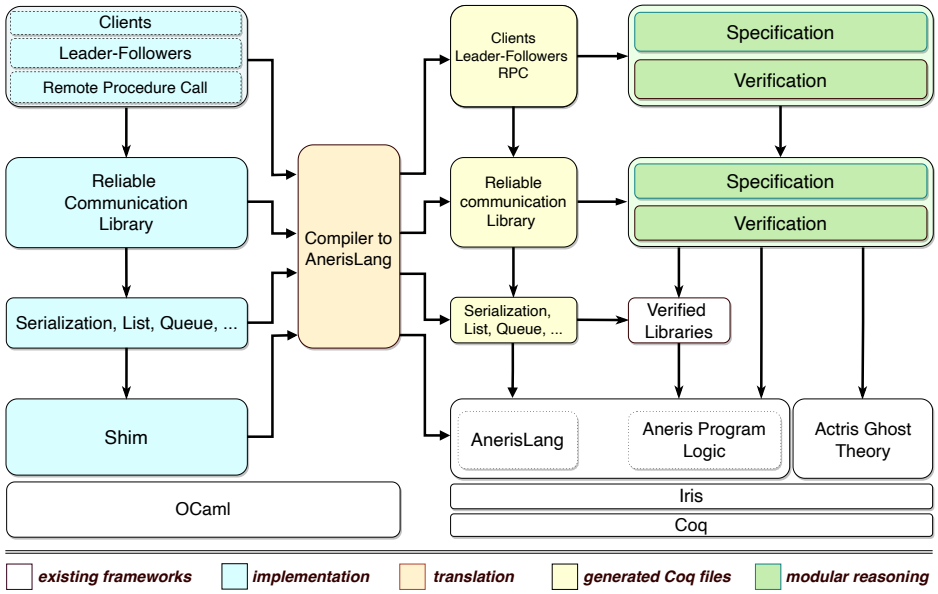

Fig. 1. The overview of our approach.

As part of this work, we have implemented a simple compiler that translates programs written in a subset of OCaml to AnerisLang, the formally defined programming language that Aneris is proved sound for. In our case, we obtain AnerisLang implementations for every OCaml implementation, as shown in the figure. This is similar to the approach taken by prior work [Chajed et al. 2019], which designed a Go-to-GooseLang compiler to reason about a subset of Go by translation to Iris.

Note however 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.

More generally, the trusted computing base of our framework 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 Iris, and by extension Aneris and Actris, are not part of the trusted computing base as their adequacy (soundness) is proven in Coq.

The core contribution of this paper is depicted by the green part of the figure and consists of the modular specification and verification of the reliable communication library and its clients. The specification and verification is done formally in Coq using the Aneris distributed separation logic, and also relies on the so-called ghost theory of Actris. Aneris is itself defined on top of the Iris base logic, which in turn is modeled and proved sound in Coq. Thus our work is what is sometimes called "foundational": the whole tower of reasoning is built on top of and within the Coq proof assistant. For closed proofs of complete programs, such as that of the leader-followers clients and server, we have used the adequacy theorems of Aneris to extract proofs in Coq that express that the verified programs are indeed safe to run w.r.t. the formal operational semantics of AnerisLang.

We leverage the fact that Aneris is defined on top of Iris to obtain highly modular and general specifications. For example, the RPC library specifies request handlers using abstract pre- and post-conditions, which can be instantiated with advanced Iris features such as higher-order concurrent abstract predicates (HOCAP) to reason about logically atomic remote procedure calls.

We stress that each component shown in the figure 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 solely 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.

Contributions. In summary, we make the following contributions:

- We present the first foundationally verified implementation of a reliable communication library for asynchronous client-server communication with FIFO at-most once message delivery guarantee (Section 2).
- We demonstrate the usefulness of our logic by verifying a generic *remote procedure call* library which can be used as a middleware component to further simplify the formal development of distributed applications (Section 3).
- On top of the RPC library we verify a *sequentially consistent 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. To the best of our knowledge, our proof is the first modular foundational verification of *sequentially consistent lazy replication* (Section 4).
- We demonstrate that the Actris framework, which has previously only been applied to message-passing concurrency, can also effectively be applied to specify and verify implementations of *distributed sessions*. To this end, we introduce a *session escrow pattern*, which conceptually merges the distributed sharing of spatial resources of Aneris with the dependent resource transfer of Actris (Section 5).
- Our specifications are proved within the Aneris framework, without changing its underlying network model or its axioms. As a result, we obtain the first program logic in which one can reason both about UDP-like communication *and* reliable client-server sessions.
- We implement a compiler that translates a subset of OCaml into AnerisLang. All of our libraries and examples are written in OCaml (~900 loc) using this compiler.

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 [Author(s) 2022].

2 RELIABLE COMMUNICATION LIBRARY API AND SPECIFICATION

In this section we present the API and specification of the reliable communication library that we have implemented and verified. We first present the API of the reliable communication library (Section 2.1). We then cover Actris, the formal foundation of our reliable communication protocol specifications (Section 2.2). Then, we present the specifications of our reliable communication library (Section 2.3), and finally, we present a simple example (Section 2.4). We return to the verification of the library itself in Section 5.

```
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. 2. The API of the reliable communication library.

2.1 Reliable Communication Library API

Figure 2 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.

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 request has been accepted by the server and the connection has been established.

Furthermore, 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, so the user can directly send and receive values of the chosen data types, instead of operating on strings, which is the standard type of message contents in Aneris. This is reflected in the API by the fact that the socket descriptor types take a pair of type parameters ('a,' b), and that in order 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 `send`, `try_recv` and `recv` functions.

2.2 Actris: specification and reasoning about reliable communication

The Actris framework [Hinrichsen et al. 2022] provides a generic means of 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 :

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

These constructors are used to specify a sequence of obligations to send (**!**) and receive (**?**), which can be terminated by **end**. More specifically, the constructors $\mathbf{!}\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\ prot$ and $\mathbf{?}\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\ prot$ 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 $!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle.\ prot$ instead of $!\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{\mathsf{True}\}.\ prot$.

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

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

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:

$$\mathtt{echo\_prot} \triangleq \mu rec.\ ?(s : \mathsf{String})\,\langle s\rangle.\,!\,(n : \mathbb{N})\,\langle n\rangle\{n = |s|\}.\ rec$$

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 a number n, which corresponds to the length of the string, as captured by the corresponding message proposition, $n = |s|$, and then recurses.

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

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

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).

To see why this 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 can guarantee that it is sound, if we can update the initially strictly dual protocol, using the protocol-preserving updates captured by the subprotocol relation, so that the dual of the echo_prot fits the client:

$$\overline{\mathtt{echo\_prot}} \sqsubseteq\ !\,(s_1 : \mathsf{String})\,\langle s_1\rangle.\,!\,(s_2 : \mathsf{String})\,\langle s_2\rangle.\\ ?(n_1 : \mathbb{N})\,\langle n_1\rangle\{n_1 = |s_1|\}.\,?(n_2 : \mathbb{N})\,\langle n_2\rangle\{n_2 = |s_2|\}.\,\overline{\mathtt{echo\_prot}}$$

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

With the dependent separation protocols in hand, we can specify our channel descriptors with the so-called channel endpoint ownership $c \overset{ip}{\underset{ser}{\rightarrowtail}} prot$, inspired by a connective of the same name from the Actris framework. The channel endpoint ownership asserts that c is a channel descriptor, of which we have exclusive ownership. It additionally captures that the channel descriptor must follow the protocol specified by *prot*, which is made formal in the following section. Finally, the channel endpoint ownership asserts that the channel endpoint lives at the node with ip address *ip*, and that values sent from it must be serializable by the serializer *ser*.

2.3 Reliable Communication API and Specifications

Similar to how the OCaml API hides the implementation details of the RCLib, our specification, shown in Figure 3, 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

RC User Parameters and Resources:

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

$$S \in \text{RC\_Resources}\ (UP : \text{RC\_UserParams}) \triangleq \left\{\begin{array}{ll} \texttt{SrvCanInit} : \text{iProp}; & \texttt{CltCanInit} : \text{Address} \to \text{iProp}; \\ \texttt{CanListen} : \text{Socket} \to \text{iProp}; & \texttt{CanConnect} : \text{Ip} \to \text{Socket} \to \text{iProp} \\ \texttt{Listens} : \text{Socket} \to \text{iProp}; & \end{array}\right\}$$

Server Setup Specifications:

HT-MAKE-SERVER-SOCKET [S]
$\{S.\texttt{SrvCanInit}\}\ \langle S.\texttt{srv.ip};\ \text{mk\_srv\_skt}\ S.\texttt{ss}\ S.\texttt{cs}\ S.\texttt{srv}\rangle\ \{w.\ \exists skt.\ w = skt * S.\texttt{CanListen}\ skt\}$

HT-LISTEN [S]
$\{S.\texttt{CanListen}\ skt\}\ \langle S.\texttt{srv.ip};\ \text{listen}\ skt\rangle\ \{S.\texttt{Listens}\ skt\}$

HT-ACCEPT [S]
$\{S.\texttt{Listens}\ skt\}\ \langle S.\texttt{srv.ip};\ \text{accept}\ skt\rangle\ \{w.\ \exists c, sa.\ w = (c, sa) * S.\texttt{Listens}\ skt\ * c \overset{S.\texttt{srv.ip}}{\underset{S.\texttt{ss}}{\rightarrowtail}} \overline{S.\texttt{prot}}\}$

Client Setup Specifications:

HT-MAKE-CLIENT-SOCKET [S]
$\{S.\texttt{CltCanInit}\ sa\}\ \langle sa.\texttt{ip};\ \text{mk\_clt\_skt}\ S.\texttt{ss}\ S.\texttt{cs}\ sa\rangle\ \{w.\ \exists skt.\ w = skt * S.\texttt{CanConnect}\ sa.\texttt{ip}\ skt\}$

HT-CONNECT [S]
$\{S.\texttt{CanConnect}\ ip\ skt\}\ \langle ip;\ \text{connect}\ skt\ S.\texttt{srv}\rangle\ \{w.\ \exists c.\ w = c * c \overset{sa.\texttt{ip}}{\underset{S.\texttt{cs}}{\rightarrowtail}} S.\texttt{prot}\}$

Reliable Data Transmission Specifications:

HT-RELIABLE-SEND
$$\left\{\begin{array}{l} c \overset{ip}{\underset{ser}{\rightarrowtail}} !\vec{x} : \vec{\tau}\ \langle v\rangle\{P\}.\ prot * \\ P[\vec{t}/\vec{x}] * \text{Ser}\ ser\ (v[\vec{t}/\vec{x}]) \end{array}\right\}\ \langle ip;\ \texttt{send}\ c\ (v[\vec{t}/\vec{x}])\rangle\ \{c \overset{ip}{\underset{ser}{\rightarrowtail}} prot[\vec{t}/\vec{x}]\}$$

HT-RELIABLE-TRY-RECV
$$\{c \overset{ip}{\underset{ser}{\rightarrowtail}} ?\vec{x} : \vec{\tau}\ \langle v\rangle\{P\}.\ prot\}\ \langle ip;\ \texttt{try\_recv}\ c\rangle\ \left\{\begin{array}{l} w.\ (w = \texttt{None} * c \overset{ip}{\underset{ser}{\rightarrowtail}} ?\vec{x} : \vec{\tau}\ \langle v\rangle\{P\}.\ prot)\ \vee \\ \quad (\exists \vec{y}.\ w = \texttt{Some}\ (v[\vec{y}/\vec{x}]) * c \overset{ip}{\underset{ser}{\rightarrowtail}} prot[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]) \end{array}\right\}$$

HT-RELIABLE-RECV
$\{c \overset{ip}{\underset{ser}{\rightarrowtail}} ?\vec{x} : \vec{\tau}\ \langle v\rangle\{P\}.\ prot\}\ \langle ip;\ \texttt{recv}\ c\rangle\ \{w.\ \exists \vec{y}.\ w = v[\vec{y}/\vec{x}] * c \overset{ip}{\underset{ser}{\rightarrowtail}} prot[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]\}$

Fig. 3. The specifications of the Reliable Communication Library

the *user parameters* (UP : RC_UserParams) provided by the user, and on the *abstract specification resources* (S : RC_Resources UP) provided by the library itself.[3] For brevity's sake, we write $S.\texttt{srv}$ as being $UP.\texttt{srv}$, whenever S : RC_Resources UP. Given a concrete instance of such user parameters, and the concrete library-provided abstract specification resources[4], the user obtains a concrete instance of the proof rules and some initial resources; here $S.\texttt{SrvCanInit}$ and $S.\texttt{CltCanInit}\ sa$ (for each client). We cover how such initial resources are freely obtained in Section 6.

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;

[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.
[4]We write iProp for the universe of Iris propositions, *e.g.* pure propositions, spatial resources, invariants, *etc.*

- *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.

The specification resources provided by the library consist of abstract predicates that the client must use to start the server and clients, and later, to set up the server and clients connection. All specifications in this paper are stated as the Hoare triples $\{P\}\ \langle ip; e\rangle\ \{x.\ Q\}$ and used for partial correctness verification of Aneris programs. Intuitively, that means that if the triple $\{P\}\ \langle ip; e\rangle\ \{x.\ Q\}$ holds, then whenever the precondition P holds, the expression e is safe to execute *on node* ip and whenever e reduces to a value v on node ip then that value should satisfy the postcondition $Q[v/x]$, where x is a binder for the return value (which we omit if the return value is unit).

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 consumes the token S.`SrvCanInit` to set up the server socket, which produces the token S.`CanListen` *skt* that must then be passed to the precondition of the Ht-listen [S] rule. In return, the postcondition of the Ht-listen [S] rule gives back to the user the token S.`Listens` *skt* which can then be passed to the precondition of the Ht-accept [S] rule in order to obtain the channel descriptor of the next incoming established connection. Note that the postcondition of the Ht-accept [S] rule both provides the user with a channel endpoint ownership $c \xrightarrow[S.\mathtt{ss}]{S.\mathtt{srv.ip}} \overline{S.\mathtt{prot}}$ for the newly created channel endpoint[5] c and gives the S.`Listens` *skt* token back (so that the accept can be called again). Note that the channel endpoint ownership has the initial protocol state $\overline{S.\mathtt{prot}}$, the dual of the user parameter protocol.

The specifications 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 turning the S.`CltCanInit` *sa* token into the S.`CanConnect` *sa*.`ip` *skt* token. The latter then allows the client to connect to the server, consuming the S.`CanConnect` *ip skt* token to produce the channel endpoint ownership $c \xrightarrow[S.\mathtt{cs}]{sa.\mathtt{ip}} S.\mathtt{prot}$. The channel endpoint ownership has the initial protocol state S.`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 Ht-reliable-send and Ht-reliable-try-recv rules.

The Ht-reliable-send rule states that to send a value, the protocol must be in a sending state $(!\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\ prot)$. We must then provide a concrete instantiation $(\vec{t}:\vec{\tau})$ of the binders $(\vec{x}:\vec{\tau})$, and give up the ownership of the resources $(P[\vec{t}/\vec{x}])$. Additionally, we must show that the value to be sent $(v[\vec{t}/\vec{x}])$ is serializable by the associated serializer *ser*. As a result, we get back the channel endpoint ownership whose protocol is updated to its dependent tail $(prot[\vec{t}/\vec{x}])$.

The Ht-reliable-try-recv rule specifies that the protocol must be in a state $(?\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\ prot)$. If there is nothing to receive, we retain ownership of the original protocol state. Otherwise, we get an instantiation $(\vec{y}:\vec{\tau})$ of the binders specified by the protocol $(\vec{x}:\vec{\tau})$, for which we obtain ownership of the resource specified by the protocol $(P[\vec{y}/\vec{x}])$, and unification of the received value (w) with the value of the protocol $(w = v[\vec{y}/\vec{x}])$. As a result, we get back the channel endpoint ownership whose protocol is updated to its dependent tail $(prot[\vec{y}/\vec{x}])$. Finally, the rule Ht-reliable-recv specifies in a similar way the receive operation (`recv`) which blocks until there is a value to return.

2.4 A Simple Example: Verifying a String Length Server

To illustrate how the RCLib specifications can be used concretely, we consider an implementation of a server that returns the length of each incoming request, as presented in Figure 4.

[5]We use "channel endpoint" and "channel descriptor" interchangeably.

```
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 a =
  let s = mk_server_skt int_ser str_ser a in
  server_listen s; accept_loop s
```

Fig. 4. Example: server returning the length of the incoming string requests.

The right-hand side of Figure 4 shows the server's code. Once the server is started and is listening to the clients on the socket `s`, it calls the accept loop. The latter returns, for each newly accepted client connection, a fresh channel descriptor c and spawns a thread on which it will serve the client `serve_loop c`. The service consists of a loop, which on each iteration receives a string as request, computes its length, and sends the result back.[6]

The left-hand side of Figure 4 shows the code for a particular client, which connects to the server's address `srv`, and, when a connection is established, acquires the channel descriptor c on which it can communicate with the server. The client then sends two consecutive messages "Carpe" and "Diem", and waits for the results $m1$ and $m2$. Note how, in order to hold, the client's assertion `assert (m1 = 5 && m2 = 4)` relies on the fact that the communication with the server is reliable.

To prove that the assertion never fails, we prove a separation logic specification for the example code and then apply the adequacy theorem (see section 6). The full formal specification and proof thereof can be found in our accompanying Coq formalization [Author(s) 2022]; 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 from Section 2.2. We thus start by instantiating the RCLib with the following user parameters:

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

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

$$\{S.\mathtt{SrvCanInit}\}\ \langle \mathtt{S.srv.ip};\ \mathtt{server}\ S.\mathtt{srv}\rangle\ \{\mathsf{False}\} \qquad (\textit{server})$$
$$\{S.\mathtt{CltCanInit}\ sa\}\ \langle sa.\mathtt{ip};\ \mathtt{client}\ sa\ \mathtt{S.srv}\rangle\ \{\mathsf{True}\} \qquad (\textit{client})$$

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 $S.\mathtt{Listens}\ skt$ 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.

[6] All examples considered in this paper follow the same multi-threaded paradigm. This is not a limitation, and we believe that our RCLib specifications also work for e.g. an event-driven paradigm, but we leave such investigation for future work.

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[7] by using the subprotocol relation as we explained in Section 2.2. The propositions of the protocol ($m1$ = |"Carpe"| and $m2$ = |"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 ∼ 350 lines of Coq code.

3 REMOTE PROCEDURE CALL LIBRARY

To demonstrate the expressivity of the RCLib specs (Section 2), we consider now the specification and verification of a multi-threaded *RPC library*. In Section 4 we will then show how this library itself is used to facilitate the formal development of clients and applications that make use of it.

A remote procedure call (RPC) service is a key middleware component of distributed systems that enables clients to call remote procedures as if the procedures were local. In RPC, the server usually exposes a set of service procedures that the clients call remotely, and those procedures (also called request handlers) can also be stateful, *i.e.* they can encapsulate the internal state of the server that the clients might wish to update remotely. RPCs can be implemented either on top of UDP or TCP, and in the latter case, the RPC benefits from the reliability guarantees.

In this work, we have implemented, specified and verified a variant of such an RPC service. This variant exposes just one service handler, but in which the types of client's request and server's response are *polymorphic* and *higher-order*. In particular, instantiating those types with sum-types $\tau_q^1 + \tau_q^2$ (for requests), and $\tau_r^1 + \tau_r^2$ (for responses) effectively allows us to encode an RPC service that handles multiple procedures 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 5 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.

3.1 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 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*.*S*.`pre` and *S*.`post`) that relate the request and reply values with their corresponding data.

In return the RPC library provides the abstract predicates (*S* : `RPC_Resources` *UP*), which consist of *S*.`CanStart`, *S*.`CanConnect` *sa*, and *S*.`CanRequest` *ip rpc* resources. The resources *S*.`CanStart` and *S*.`CanConnect` *sa* govern the permission to start the server and allow clients to connect to it, respectively, and, similar to the RCLib, are freely obtained at the initial setup (again, see Section 6).

To start the RPC service the user must provide the precondition of the Ht-rpc-start [S] specification, the *S*.`CanStart` token, and the proof that the procedure *proc* satisfies the specification defined by `rpc_process_spec`. Indeed, this specification ensures the procedure function handles

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

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:

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

$$S \in \mathsf{RPC\_Resources}\ (UP : \mathsf{RPC\_UserParams}) \triangleq \left\{\mathsf{CanStart} : \mathsf{iProp};\quad \mathsf{CanConnect} : \mathsf{Address} \to \mathsf{iProp};\quad \mathsf{CanRequest} : \mathsf{Ip} \to \mathsf{Val} \to \mathsf{iProp}\right\}$$

RPC Specifications:

HT-RPC-CONNECT [S]

$$\{S.\mathsf{CanConnect}\ sa\}\ \langle sa.\mathsf{ip};\ \mathsf{rpc\_connect}\ S.\mathsf{qs}\ S.\mathsf{rs}\ sa\ S.\mathsf{srv}\rangle\ \{rpc.\ S.\mathsf{CanRequest}\ sa.\mathsf{ip}\ rpc\}$$

HT-RPC-START [S]

$$\{S.\mathsf{CanStart} * \mathsf{rpc\_process\_spec}\ S\ proc\}\ \langle S.\mathsf{srv.ip};\ \mathsf{rpc\_start}\ S.\mathsf{rs}\ S.\mathsf{qs}\ S.\mathsf{srv}\ proc\rangle\ \{\mathsf{True}\}$$

HT-RPC-REQUEST [S]

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

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

Fig. 5. Specifications for the RPC library.

the incoming requests correctly. In particular, `rpc_process_spec` states that the procedure argument qv must satisfy the provided precondition $S.\mathsf{pre}\ qv\ qd$, and that the results rv must satisfy the provided postcondition $S.\mathsf{post}\ rv\ qd\ rd$. As such, it is thus necessary for the user to prove `rpc_process_spec`, for the procedure function that they choose, when starting the server.

The $S.\mathsf{CanConnect}\ sa$ resource is used once per client to connect to the RPC service on the given socket address, which in turn yields the $S.\mathsf{CanRequest}\ ip\ rpc$ resource for the returned RPC handle rpc, as specified by HT-RPC-CONNECT [S]. Finally, the HT-RPC-REQUEST [S] specification captures how the client can make requests when in possession of the $S.\mathsf{CanRequest}\ ip\ rpc$ resource. Additionally, the argument qv must satisfy the provided precondition $S.\mathsf{pre}\ qv\ qd$, and qv must be serializable by the provided request serializer $S.\mathsf{qs}$. In return the client obtains the resources of the postcondition $S.\mathsf{post}\ rv\ qd\ rd$ for the returned value rv.

3.2 Verification of the RPC library

The verification of the RPC library primarily boils down to showing that the specification of the client's `rpc_make_request` follows from the user provided proof of the request handler at the server side w.r.t the specification `rpc_process_spec`. The crux of this connection is to come up with a dependent separation protocol that specifies the delegation of the handler call to the server:

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

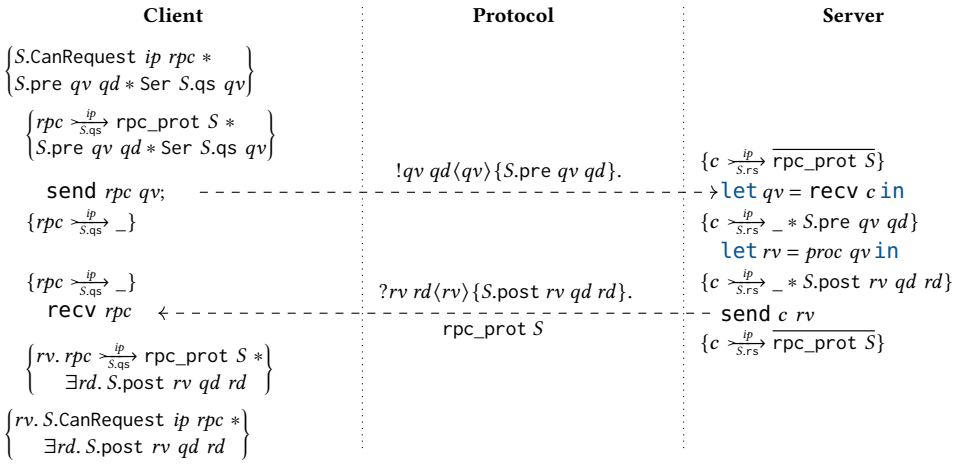

Fig. 6. The reliable communication of the RPC library

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*.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*.post *rv qd rd* predicate.

Figure 6 sketches the proof of how this protocol connects the client's local and remote calls specifications to verify Ht-rpc-request [S]. First the abstract resource *S*.CanRequest *ip rpc* is unfolded, to obtain the channel endpoint ownership $rpc \overset{ip}{\underset{S.\mathsf{qs}}{\rightarrowtail}} (\mathtt{rpc\_prot}\ S)$. The resources for the request value (*S*.pre *qv qd*) are then 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*.post *rv qd rd*, which are then sent back to the client. 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*.CanRequest *ip rpc*, so that postcondition of the rpc_make_request holds, thus concluding the proof.

4 SEQUENTIALLY CONSISTENT LAZY REPLICATION WITH LEADER-FOLLOWERS

It is well-known that due to the CAP theorem [Gilbert and Lynch 2002] online services, cannot at the same time have the three important properties of consistency (all replicas agreeing on the state of the system at all times), availability (being responsive in a timely fashion), and partition-tolerance (functioning in the presence of network failures). Hence, online services often try to strike a balance between these properties. depending on the application at hand.

The system that we present in this section is a replicated KVS with different guarantees for read and write operations. The entire system, *i.e.*, the leader and all the followers as we will explain, is guaranteed to agree upon, and preserve, the order of write operations. This is achieved by having a central server node, called the *leader*, which registers all the write operations. The state of the leader is then lazily replicated by so-called *follower* servers which periodically poll the state of the leader and store a local copy. The idea is that a client has to direct all the write requests to the leader while they have a choice to direct read operations at the leader or any of the followers. The read operation directed at the leader is guaranteed to always return the most up-to-date value while those directed at a follower may return a stale value.

4.1 Specification for the Leader and Followers

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
$\{k \mapsto^{\mathsf{ldr}} vo\}\ \langle ip;\ write\ k\ v\rangle\ \{k \mapsto^{\mathsf{ldr}} \mathsf{Some}\ v\}$

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

Here the $k \mapsto^{\mathsf{ldr}} vo$ proposition where vo is an optional value (similar to the usual points-to proposition in standard separation logic except instead of values we use optional values) 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 (Indeed the spec LEADER-ONLY-WRITE-SPEC, as we will discuss below, can be derived from our general spec for the write operation given in Figure 7). 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 $Obs(\mathrm{DB}, h)$, defined in terms of Iris resources, to indicate that the history h (a sequence of write events) 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 are persistent, *i.e.*, $Obs(\mathrm{DB}, h) \dashv\vdash Obs(\mathrm{DB}, h) * Obs(\mathrm{DB}, h)$, and only express the knowledge that a certain history has been observed. 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. This, as we will see in Section 4.2, 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 $k \mapsto^{\mathsf{ldr}} v$, the logical state of what is observed by each server, exposed to the client as $Obs(\mathrm{DB}, h)$, and the physical state (stored in the memory) of each server which is not exposed to the client. The following tables give a summary of the building blocks used in the specification of 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.
$Obs(\mathrm{DB}, h)$	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 $Obs(\mathrm{DB}, h)$ 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.

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

LEADER-READ-SPEC
$$\left\{k \mapsto^{\text{kvs}}_{q} wo\right\} \langle ip;\ read\ k\rangle \left\{x.\ k \mapsto^{\text{kvs}}_{q} wo\ *\ ((x = \text{None} \land wo = \text{None}) \lor (\exists we.\ x = \text{Some } we.value \land wo = \text{Some } we))\right\}$$

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

Fig. 7. The specification for the write operation and the read operation for both the leader and followers.

These are the important properties of observations:

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

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

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

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 7. 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 $k \mapsto^{\text{kvs}} wo$, 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 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) .

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:

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

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

```
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. 8. Example Client of Leader-Followers.

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 $k \mapsto^{\text{kvs}} wo$, which we get by unfolding the definition of $k \mapsto^{\text{ldr}} vo$, and a case analysis on whether vo is None or Some v.

4.2 Client Example

The Figure 8 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 KVS, *i.e.*, the leader and the followers, have been initialized prior to running these clients.

One client, `client0`, only performs two write operations, 37 to x followed by 1 to y. The other client, `client1`, only reads, and the read operation is directed at a follower. `client1` 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.

The example above demonstrates that reading from a follower satisfies three out of the four so-called session guarantees [Terry et al. 1994], *i.e.*, the salient properties of causal consistency, namely *monotonic reads*, *monotonic writes*, and *writes follow reads*. Note that the leader does not synchronize with followers during the write operation. Hence, the KVS does not provide the *read-your-writes* guarantee when the read operation subsequent to a write operation is directed at a follower. The essence of showing correctness of the example in Figure 8 amounts to showing that the assertion in `do_reads` function does not fail. To prove this, we use an Iris invariant together with points-to propositions and the leader's observations (similar to how they are tied together in the pre- and postcondition of WRITE-SPEC) to assert that at all times there is at most one write operation on x and at most one write operation on y and the former happens before the latter.

4.3 Implementation and Verification

So far we described how we specify the leader-followers KVS and how the client's code can use those specifications. Here we give a brief overview of how we implement the leader and followers, and how we verify them w.r.t. the specification presented above.

The KVS, *i.e.*, both the leader and followers, is implemented directly on top of the RPC library. That is, we only implement handlers which, upon clients' requests, write (at the leader) or read (at the leader or follower) the local state of the server. The local state 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. [8] The crux of the verification is then to:

- Give concrete definitions of the abstract predicates, *e.g.*, $Obs(\mathsf{DB}, h)$ and $k \mapsto^{\mathsf{kvs}} wo$.
- Instantiate the specifications of the RPC library for handlers.
- Show the Hoare triples for the handlers as ascribed by the RPC library.

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 9. 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:[9]

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

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

[8] Technically in the implementation we use monitors which are very similar to locks except in that instead of busy waiting they put the thread to sleep. From a verification point of view though locks and monitors are fairly similar and hence not worth discussing in detail here.
[9] The local invariant is essentially stated as a lock invariant. See [Birkedal and Bizjak 2017] for locks in Iris.

Table-lookup
$$KWT\,(M) * k \mapsto^{\mathsf{kvs}} wo \vdash M(k) = wo$$

Table-update
$$KWT\,(M) * k \mapsto^{\mathsf{kvs}} wo \Rrightarrow KWT\left(M[k \mapsto wo']\right) * k \mapsto^{\mathsf{kvs}} wo'$$

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

Obs-prefix
$$\frac{X \in \{\mathsf{G}, \mathsf{S}, \mathsf{L}\}}{Log_X(\mathsf{DB}, h) * Obs(\mathsf{DB}, h') \vdash h' \leq_p h}$$

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

Fig. 9. Rules governing the internal leader-followers library propositions.

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 $\ell_{log_{\mathrm{DB}}}$.

Given these propositions and invariants we instantiate the RPC library by taking the precondition and the postcondition of the handler to be the combination of the preconditions and postconditions of the read and write operation; the RPC request is essentially a tagged request specifying whether the request is read or write along with the relevant data. One nuance that we have avoided is that the specs that we have given to the read and write operations do not take advantage of the fact that these operations are logically atomic. A logically atomic operation is an operation that is not physically atomic, *i.e.*, in the small-step operational semantics it takes more than a single step, but still effectively behaves atomically. Our general specs for the read and write operation follow the so-called HOCAP-style of specifications which allow us to take advantage of the logical atomicity of these operations, *i.e.*, we can open invariants around these operations as though they were physically atomic. (In particular, opening invariants around read and write operations is needed for the proof of the causality example.) The specs we presented earlier are weaker than (can be derived from) the HOCAP-style specs. That being said, given ordinary specs for a logically atomic operation it is rather easy to come up with the corresponding HOCAP-style specification. See [Gondelman et al. 2021] for a discussion on HOCAP-style specs and our formal Coq development for more details of how they are used, *e.g.*, in the proof of the causality example presented above.

Showing that the Hoare triples for the handler functions as ascribed by the specification of the RPC library is rather straightforward. We only need to show that during the three main operations of the KVS, *i.e.*, reading, writing, and updating the follower, the local and global invariants are preserved. Note that in reasoning about these simple properties we do not need to reason about the UDP network (handled by the reliable communication library), or the communication protocol used (handled by the RPC library). These properties simply follow from the rules governing the abstract predicates we presented earlier. This shows the power and flexibility of our approach and the idea of vertical modularity, *i.e.*, the idea that libraries are separate modules verified separately.

5 VERIFICATION OF THE RELIABLE COMMUNICATION LIBRARY

We have so far presented the specification of the RCLib and how to use it for the verification of clients and libraries on top of it. In this section, we provide insight on how we implement and verify the key part of the RCLib—the *channel descriptors*—w.r.t. the specifications given in Figure 3. We

$$\mathsf{True} \Rrightarrow \exists \chi.\, \mathsf{prot\_ctx}\ \chi\ \epsilon\ \epsilon * \mathsf{prot\_own_l}\ \chi\ \mathit{prot} * \mathsf{prot\_own_r}\ \chi\ \overline{\mathit{prot}} \tag{PROTO-ALLOC}$$

$$\mathsf{prot\_ctx}\ \chi\ \vec{v_1}\ \vec{v_2} * \mathsf{prot\_own_l}\ \chi\ (!\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) * P[\vec{t}/\vec{x}] \Rrightarrow \left(\triangleright^{|\vec{v_2}|}\ \mathsf{prot\_ctx}\ \chi\ (\vec{v_1} \cdot [v[\vec{t}/\vec{x}]])\ \vec{v_2}\right) * \mathsf{prot\_own_l}\ \chi\ (\mathit{prot}[\vec{t}/\vec{x}]) \tag{PROTO-SEND-L}$$

$$\mathsf{prot\_ctx}\ \chi\ \vec{v_1}\ ([w] \cdot \vec{v_2}) * \mathsf{prot\_own_l}\ \chi\ (?\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}) \Rrightarrow \triangleright\, \exists \vec{y}.\, (w = v[\vec{y}/\vec{x}]) * P[\vec{y}/\vec{x}] * \mathsf{prot\_ctx}\ \chi\ \vec{v_1}\ \vec{v_2} * \mathsf{prot\_own_l}\ \chi\ \mathit{prot}[\vec{y}/\vec{x}] \tag{PROTO-RECV-L}$$

Fig. 10. Excerpt of rules of the Actris ghost theory.

focus on how we achieve the reliable transfer of the dependent resources specified by the dependent protocols via a novel proof pattern—the *session escrow pattern*—which conceptually merges the distributed sharing of spatial resources of Aneris with the dependent resource transfer of Actris. We first explain how we implement the channel descriptors of the RCLib (Section 5.1). We then cover the session escrow pattern, and how it resolves key limitations of using Aneris and Actris for reliable distributed transfer (Section 5.2). 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 5.3).

5.1 Implementation of the RCLib Channel Descriptors

The main component of the RClib implementation is a *channel descriptor*; a local physical state that is mutated both by user calls to the send/receive operations and the internal protocol procedures that enable reliable data transmission.

Concretely, a channel descriptor is implemented as a 5-tuple $(\ell_{sbuf}, \ell_{rbuf}, \mathit{ser}, \mathit{slk}, \mathit{rlk})$. The send buffer ℓ_{sbuf} is a reference to a pair $(\mathit{sbuf}, \mathit{sid})$ where sbuf is an outbound queue storing pairs of values and sequence ids, and where sid is a sequence identifier designating the next sequence number n. Each call to `send` $c\ v$ then pushes a pair (v, n) to sbuf and increments the value of sid. In parallel to user calls, the internal sending procedure (a non-terminating loop) then serializes each pair using the ser serialization and transmits it to the other session endpoint via the network, until a delivery acknowledgement is received, after which it is removed from the sbuf queue. To this effect, the internal procedure on each side maintains an additional reference ℓ_{sidLB} to a counter that keeps track of the lowest sequence id currently stored in the sbuf (so that $\mathit{sid} = \mathit{sidLB} + |\mathit{sbuf}|$).

The reference ℓ_{rbuf} stores a receive buffer rbuf implemented as a queue containing values coming from the other session endpoint; the internal receiving procedure (again a non-terminating loop) pushes each incoming value to the rbuf exactly once and in the order which those values have been sent. To this effect, it uses an additional reference ℓ_{ackid} as a counter that is used to check whether an incoming message should be added to rbuf. Each user's call to `recv` c then simply pops the value at the head of 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 internal procedures and user's calls to 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, which simplifies the verification as well. All the networking and reliability checks are delegated to the implementation of internal procedures, which we do not detail here.

5.2 Modelling Reliable Transfer with the Actris Ghost Theory and Message Histories

To ensure the reliable transfer of the resources described by the dependent protocols we draw inspiration from a pattern used in Aneris, where safe transfer of spatial resources over the unreliable network is achieved by storing the spatial resources in a shared logical context, and then sending a

AUTH-LIST-ALLOC
$\text{True} \Rrightarrow \exists \gamma.\, \text{auth\_list}\ \gamma\ \epsilon * \text{list\_len}\ \gamma\ 0$

AUTH-LIST-EXTEND
$\text{auth\_list}\ \gamma\ \vec{x} * \text{list\_len}\ \gamma\ n \Rrightarrow \text{auth\_list}\ \gamma\ (x \cdot [\vec{x}]) * \text{list\_len}\ \gamma\ (n+1) * \text{frag\_list}\ \gamma\ n\ x$

AUTH-LIST-AGREE
$$\frac{\text{auth\_list}\ \gamma\ \vec{x} \quad \text{frag\_list}\ \gamma\ i\ x}{\vec{x}_i = x}$$

AUTH-LIST-LENGTH
$$\frac{\text{auth\_list}\ \gamma\ \vec{x} \quad \text{list\_len}\ \gamma\ n}{|\vec{x}| = n}$$

FRAG-LIST-DUP
$$\frac{\text{frag\_list}\ \gamma\ i\ x}{\text{frag\_list}\ \gamma\ i\ x * \text{frag\_list}\ \gamma\ i\ x}$$

Fig. 11. The monotonic list ghost theory.

duplicable witness over the network.[10] This effectively enables retransmission (as the witness is duplicable), and safe transfer (as the spatial resources can only be taken out once). However, there are two constraints that we need to address to use this pattern for dependent sessions:

(c1) The pattern as-is does not allow dependencies between the resources stored in the shared logical context (indeed, there might be several resources *in transit*). We thus need a non-trivial mechanism for releasing to, storing in, and acquiring from the shared logical context.

(c2) The duplicable witnesses must appropriately reflect the state of such a mechanism, so that resources can be acquired in accordance to their dependence.

Actris ghost theory. Constraint (c1) can be addressed by the Actris ghost theory [Hinrichsen et al. 2022], the relevant fragment of which is given in Figure 10. In this ghost theory, the dependent transfer of resources is modelled by using two lists $\vec{v_1}, \vec{v_2}$ describing the *messages in transit* in each direction, and three resources, namely $\text{prot\_ctx}\ \chi\ \vec{v_1}\ \vec{v_2}$, $\text{prot\_own}_\text{l}\ \chi\ \mathit{prot}$, and $\text{prot\_own}_\text{r}\ \chi\ \overline{\mathit{prot}}$. The intuition is that $\text{prot\_ctx}\ \chi\ \vec{v_1}\ \vec{v_2}$ represents the authority over the resources of the messages that are in transit, and it is the key component for defining the shared logical context.[11] The resources $\text{prot\_own}_\text{l}\ \chi\ \mathit{prot}$ and $\text{prot\_own}_\text{r}\ \chi\ \overline{\mathit{prot}}$ represent the current state of the shared logical context from the perspective of the left and right endpoint, respectively, and are the key components for defining the channel endpoint connective $c \rightarrowtail^{\substack{ip \\ ser}} !\,\vec{x}{:}\vec{\tau}\,\langle v\rangle\{P\}.\,\mathit{prot}$. The rules in Figure 10 show how to allocate those resources (PROTO-ALLOC), and how to resolve the steps of releasing (PROTO-SEND-L) and acquiring (PROTO-RECV-L) the resource P, updating the logical state of the protocol accordingly. We omit the rules about the transfer from right to left, as they are symmetric. Finally, the conclusion of the rules are guarded by the later modality $\triangleright$. While these carry significance, we defer a discussion of them to the end of this section, and ask the reader to disregard them for now.

Histories of sent and received messages. To address constraint (c2) we need to connect the state of the Actris model with some persistent witnesses that reflect the state of the logical buffers of $\text{prot\_ctx}\ \chi\ \vec{v_1}\ \vec{v_2}$. We observe that this can achieved with four message histories Tl, Tr, Rl, and Rr, that keep track of all the messages transferred and received so far by the left and right session endpoints, respectively. To this end we employ a ghost theory of histories (monotonically growing lists), as presented in Figure 11. Intuitively this ghost theory captures the authoritative state of a list (via $\text{auth\_list}\ \gamma\ \vec{x}$), and its length (via $\text{list\_len}\ \gamma\ n$), and persistent witnesses of the values of its individual indices (via $\text{frag\_list}\ \gamma\ i\ x$). The rules reflect how the list can intuitively be allocated and extended, and how the authoritative list, length fragment, and witnesses agree with each other.

The shared logical context can then be captured as the following Iris invariant:

$$\exists \mathit{Tl}, \mathit{Tr}, \mathit{Rl}, \mathit{Rr}.\, \text{auth\_list}\ \chi_{\mathsf{Tl}}\ \mathit{Tl} * \text{auth\_list}\ \chi_{\mathsf{Tr}}\ \mathit{Tr} * \text{auth\_list}\ \chi_{\mathsf{Rl}}\ \mathit{Rl} * \text{auth\_list}\ \chi_{\mathsf{Rr}}\ \mathit{Rr} * \text{prot\_ctx}\ \chi_{\mathsf{chan}}\ (\mathit{Tl} - \mathit{Rr})\ (\mathit{Tr} - \mathit{Rl}) * \mathit{Rr} \leq_\text{p} \mathit{Tl} * \mathit{Rl} \leq_\text{p} \mathit{Tr} * \boxtimes\, |\mathit{Tl}| * \boxtimes\, |\mathit{Tr}|$$

[10] This pattern is an instance of the general ownership transfer mechanism called *escrows* introduced by Kaiser et al. [2017].
[11] Here χ, χ_{Rr}, χ_{Tl}, γ, ... are a so-called *ghost* identifiers that relate the ownership of resources in the logic in Iris.

$$\textsc{Ht-step-get}\ \frac{\{P * ⧖\,0\}\ \langle ip;\ e\rangle\ \{\Phi\}}{\{P\}\ \langle ip;\ e\rangle\ \{\Phi\}} \qquad \textsc{Ht-step-incr}\ \frac{\{P\}\ \langle ip;\ e\rangle\ \{w.\,Q\}}{\{P * ⧖\,n\}\ \langle ip;\ e\rangle\ \{w.\,Q * ⧖\,n+1\}} \qquad \textsc{Ht-step-frame}\ \frac{\{P\}\ \langle ip;\ e\rangle\ \{w.\,Q\}}{\{P * ⧖\,n * \triangleright^n R\}\ \langle ip;\ e\rangle\ \{w.\,Q * R\}} \qquad \textsc{step-dup}\ \frac{⧖\,n}{⧖\,n * ⧖\,n}$$

Fig. 12. The mechanism for stripping multiple laters. We require e to be an atomic expression.

The invariant asserts the authoritative ownership of the message histories (Tl, Tr, Rl, and Rr) and of the shared logical state in terms of the histories prot_ctx χ_{chan} $(Tl - Rr)$ $(Tr - Rl)$. It additionally captures the prefix relations $Rr \leq_{\mathsf{p}} Tl$, and $Rl \leq_{\mathsf{p}} Tr$ between the message histories. Finally, it owns the $⧖\,|Tl|$ and $⧖\,|Tr|$ fragments which are part of the mechanism for stripping the laters of the Actris ghost theory rules, which we describe below.

Stripping multiple laters. In Iris, and thus Aneris, one can strip a later whenever a step of computation is taken. Conventionally the intuition is that one step equates stripping one later. However, recent discoveries [Matsushita et al. 2022; Mével et al. 2019; Spies et al. 2022] uncovered various methods for stripping *mutliple* laters per step. Based on these discoveries we extended Aneris with a similar, albeit more simplistic, mechanism as presented in Figure 12. The mechanism lets us strip multiple laters during one physical step, based on the amount of steps that has been taken thus far. The rule Ht-step-get lets us track a new lower bound of steps taken thus far $⧖\,0$, and Ht-step-incr allows us to increase it by one, every time a step is taken. Crucially, the rule Ht-step-frame lets us frame resources under an amount of laters corresponding to the lower bound of steps taken thus far, as signified by $⧖\,n$. The session invariant, presented above, tracks the lower bounds $⧖\,|Tl|$ and $⧖\,|Tr|$, which lets us compensate the steps incurred by the Actris ghost theory rules. It is worth pointing out that this approach to stripping multiple laters improves upon the original application of the Actris ghost theory, which instead employed course-grained concurrency using a physical lock to retain ownership of the resources until all the necessary steps were taken.

5.3 Proof of the RCLib Implementation

With the generic model of the shared logical context presented in Section 5.1, we can now give an overview of how we verify the network-agnostic part of the RCLib. To do so, we must (1) tie the shared logical context to the physical state of the channel descriptors, and (2) tie the channel descriptors to the in- and outbound physical buffers. We achieve both by constructing the channel endpoint ownership $c \overset{ip}{\underset{ser}{\rightarrowtail}} prot$. For the sake of brevity, we focus on the left channel endpoint, governing Tl and Rl. The definition governs the (duplicable) ownership of the session invariant, along with the length fragments of Tl and Rl. The length of Tl is tied to the physical sequence id sid, so that outbound messages are required to be fresh and in order. Similarly, the length of Rl is tied to the value stored in the acknowledgments reference ℓ_{ackid} to guarantee that received messages are fresh and in order. Finally, the channel endpoint ownership governs the lock invariants of the outbound and inbound buffers, that each govern duplicable witnesses frag_list $\gamma\ i\ v$ of the related entries in their buffers. For instance, the i-th value v currently stored in the receive buffer $rbuf$ of a channel's left endpoint carries a duplicable witness frag_list χ_{Tr} $(ackid + i)$ v, that is a witness generated by the other session endpoint and relating its sending history Tr.

Proof sketch for the send and receive operations. With the logical model in place, it is relatively straightforward to verify the send and receive operations, as the channel endpoint ownership guarantees that the send and receive buffers and current sequence id corresponds to the logical message history of transmitted and received messages.

The verification of HT-RELIABLE-SEND is carried out by resolving the physical steps as described in Section 5.1. The proof follows by (1) adding the value v and resources P to the shared logical context using PROTO-SEND-L, (2) recovering the current sequence id n from the physical state, (3) extending the history of transmitted messages auth_list χ_{Tl} Tl with the v (now pushed to the send buffer along with n) to synchronise with the updated protocol fragment, and to obtain the duplicable witness frag_list χ_{Tl} n v, (4) closing the send lock by giving it the duplicable witness frag_list χ_{Tl} n v.

The verification of HT-RELIABLE-RECV is carried out by resolving the physical steps as described in Section 5.1. The proof follows by (1) closing the receive lock by taking out the duplicable witness frag_list χ_{Tr} i v, (2) deriving that v must be the first value in $Tr - Rl$, (3) obtaining the resources P from the shared logical context using PROTO-RECV-L, (4) extending the history of received messages auth_list χ_{Rl} Rl with the received value to synchronise with the updated protocol fragment.

Establishing a distributed session. While established sessions have disjoint resources, the resources are initially allocated together (e.g. using the PROTO-ALLOC). This happens on the server side, during the handshake, after which the server allocates the session invariant, and transfers the client's resources (e.g. the $\text{prot\_own}_{\text{I}}$ χ $prot$ fragment) to the client. To facilitate this, the client and server must agree on the session protocol before the handshake, which is covered as a part of the RCLib specifications. 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.

6 ADEQUACY: OBTAINING THE INITIAL RESOURCES AND CLOSED PROOFS

The component specifications presented in the paper depend on resources that we have claimed to be obtained for free at the start of a verification. This is sound because closed proofs of complete programs in Aneris are instantiated with a concrete network configuration, for which initial resources are provided which we can use to derive the component-specific resources. This is formally captured by the foundationally mechanised Aneris adequacy theorem:

THEOREM 6.1 (ADEQUACY OF ANERIS). *Let $\varphi \in \mathsf{Val} \to \mathsf{Prop}$ be a meta-level (i.e. Coq) predicate over values and suppose that the following is derivable in Aneris:*

$$\exists (cfg : \mathsf{Ip} \xrightarrow{fin} \mathsf{Set\ Port}).\ ip \notin (\mathsf{dom}\ cfg) * \{\mathsf{NetRes}\ cfg\}\ \langle ip;\ e\rangle\ \{\varphi\}$$

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.*

The first step is to pick the network configuration ($cfg : \mathsf{Ip} \xrightarrow{\text{fin}} \mathsf{Set\ Port}$), consisting of the network node ips (excluding the ip of the initial node), and their ports. We must then prove the Hoare triple, in which we start with the initial network resources NetRes cfg (left abstract for brevity's sake).

With the initial network resources in hand, we just need to derive the initial component-specific resources from them. We achieve this with a component-specific rule, that is proven as a part of each component library. As an example, the rule for the RCLib component is as follows:

$$\mathsf{NetRes}\ \left(cfg \cup \{[S.\mathtt{srv.ip} := \{S.\mathtt{srv.port}\}]\} \cup \biguplus_{sa \in sas} \{[sa.\mathtt{ip} := \{sa.\mathtt{port}\}]\}\right) \Rrightarrow \\ \mathsf{NetRes}\ cfg * S.\mathtt{SrvCanInit} * (\mathbin{\ast}_{sa \in sas} S.\mathtt{CltCanInit}\ sa) * \langle \text{component-specific specs} \rangle$$

The rule captures how we can extract the component-specific server and client network resources and convert them into the initial server and client tokens ($S.\mathtt{SrvCanInit}$ and $\mathbin{\ast}_{sa \in sas} S.\mathtt{CltCanInit}\ sa$), along with the component-specific program specifications, hidden for brevity.

Finally, to obtain a closed proof, we must consider how network nodes are started. Aneris initialises a network of nodes through a so-called *system* node, which has elevated permissions to start new nodes. An instance of such a system node would be the one for the example of Section 2.4:
`let system = start (srv_sa.ip) (server srv_sa); start (clt_sa.ip) (client clt_sa srv_sa)`

Here `srv_sa` and `clt_sa` are some concrete disjoint socket addresses. To verify such a system node, we make use of the following Aneris rule for starting the individual nodes:

$$\textsc{Ht-start}\quad \frac{ip \in \mathrm{dom}\ cfg \qquad ip \neq ip_{\mathsf{sys}} \qquad \{P\}\,\langle ip;\, e\rangle\,\{w.\,\mathsf{True}\}}{\{P * \mathsf{NetRes}\ cfg\}\,\langle ip_{\mathsf{sys}};\, \mathtt{start}\ ip\ e\rangle\,\{\mathsf{NetRes}\ (cfg \setminus ip)\}}$$

This rule states that we can start a new node, provided that we have permission to start a node on the target ip (captured by NetRes *cfg* where $ip \in \mathrm{dom}\ cfg$), and that the target ip node is different from the system ip ($ip \neq ip_{\mathsf{sys}}$). As a result, the permission to start another node on the target ip is given up (NetRes $(cfg \setminus ip)$). We must additionally verify the node, given some resources P. For the server and client of the example we would choose S.`SrvCanInit` and S.`CltCanInit clt_sa`, respectively, which we extracted from the network resources, as detailed above.

7 RELATED WORK

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.

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 library on top of Aneris's UDP primitives which they use to implement conflict-free replicated data types (CRDTs). Their implementation uses timestamps as sequence ids to achieve causal reliable delivery of broadcast UDP messages and focuses on applications that are more suited for symmetric group communication *e.g.*, CRDTs.

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, the design of this transformer, stated in a domain- specific event-handler language, is specific to the Verdi methodology. In contrast, the RCLib we present in this work is a realistic OCaml implementation of a reliable transport communication layer *à la* SCTP.

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].

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.

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. We thus achieve full *vertical modularity i.e.* the libraries are separate modules verified separately. 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 regarding realism and conventional guarantees.

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.

Currently, 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 currently 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 initial connection step includes a secure handshake, after which the connection is provably secure.

ACKNOWLEDGMENTS

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research ins Program Verification (CPV), from the VILLUM Foundation. During parts of this project Amin Timany was a postdoctoral fellow of the Flemish research fund (FWO).

REFERENCES

Anonymous Author(s). 2022. Supplementary material.

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

Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News* 33, 2 (jun 2002), 51–59. https://doi.org/10.1145/564585.564601

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

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. 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, 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

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