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, including a distributed lock manager and 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 each 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 (Berkely 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.

2023. 2475-1421/2023/1-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 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, in this paper, 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 primitives for unreliable network communication and the verification of the implementation leverages Aneris' facilities for reasoning about UDP-like unreliable communication primitives.[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 simply *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. 2020] 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 a distributed lock manager, an RPC service, and

[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.

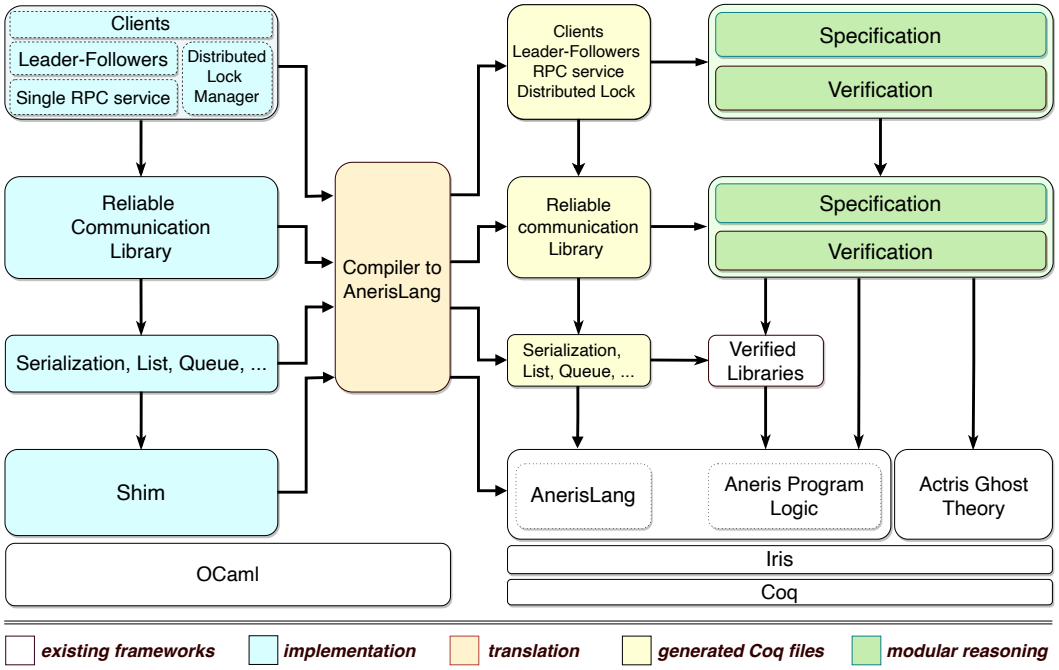

Fig. 1. The overview of our approach.

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.

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. The formal operational semantics of AnerisLang matches, by design, the intended operational semantics for the subset of OCaml that we use (Note that there is no official formal semantics for OCaml). Thus we obtain AnerisLang implementations for every OCaml implementation, as shown in the figure.

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 foundational: the whole tower of reasoning is built on top of and within the Coq proof assistant. For closed examples, such as the client programs of the leader-followers, 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, our distributed lock manager can be used to guard any Iris resource, just like a regular concurrent lock module in Iris can. Similarly, the RPC library specifies request handlers using abstract pre- and post-condition, 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 implementation details of the libraries. 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 function. In particular, the verification of the leader-followers implementation 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 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* (Section 2).
- We demonstrate the usefulness of our logic by verifying several examples such as a *distributed lock manager*, which allow client reasoning about atomic transactions. Furthermore, we develop 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).
- Using 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 leaders writes. To the best of our knowledge, our proof is the first modular foundational verification of *sequentially consistent lazy replication* (Section 4).
- We prove the specifications for the reliable communication library 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 (Section 5).
- 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 Aneris logic and Actris framework in the Coq proof assistant. The development is available in the accompanying artifact [Author(s) 2022].

2 RELIABLE COMMUNICATION LIBRARY

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.

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.

```
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.

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 aforementioned socket descriptor types take a pair of type parameters ($'$a,$'$ b), and that in order to create a client or server socket, 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. 2020] provides a generic means of specifying and reasoning about reliable communication for arbitrary implementations of reliable communication.[3] 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

[3] In this section we only cover the details of the Actris framework that are relevant to the this paper. For a more detailed presentation, see Hinrichsen et al. [2020].

the exchanges that was 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 sound communication, since whenever one endpoint expects some message, the other endpoint will send just that, and vice versa.

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

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

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, and therefore safe to perform without coordination. The most prominent such protocol-preserving update is that of *swapping*, formally captured by the following relation:

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

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:

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

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 $c \rightarrowtail^{ip}_{ser} prot$, 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 reliable communication library, our specification, shown in Figure 3, hides the details of how the implementation is verified that are irrelevant to the user. It does so by using a so-called *dependent specification pattern*, in which the specifications of the API primitives are dependent on the *user parameters* (UP : RC_UserParams)

RC User Parameters and Resources:

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

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

Server Setup Specifications:

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

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

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

Client Setup Specifications:

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

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

Reliable Data Transmission Specifications:

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

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

Fig. 3. The specifications of the Reliable Communication Library

provided by the user, and on the *abstract specification resources* (S : RC_Resources UP) provided by the library itself.[4] For brevity's sake, we write e.g. $S.\texttt{srv}$ 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, 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 5.2. We now explain each of those three components (user parameters, abstract specification resources, and specifications of the API primitives).

To initialize the library, the user must supply the following four parameters:

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

[4]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.

The specification resources that the library then provides 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, as described in detail below.

Server 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 $S.\texttt{SrvCanInit}$ to set up the server socket, which produces the token $S.\texttt{CanListen}\ skt$ that must then be passed to the precondition of the Ht-listen [S] rule, in order to put the server into listening mode. In return, the postcondition of the Ht-listen [S] rule gives back to the user the token $S.\texttt{Listens}\ skt$ 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 not only provides the user with a channel endpoint ownership $c \succ^{\frac{S.\texttt{srv.ip}}{S.\texttt{ss}}} \overline{S.\texttt{prot}}$ for the newly created channel endpoint[5] c but also gives the $S.\texttt{Listens}\ skt$ token back (so that the accept function can be called again). Finally, note that the channel endpoint ownership has the initial protocol state $\overline{S.\texttt{prot}}$, the dual of the user parameter protocol.

Client setup specifications. 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 $S.\texttt{CltCanInit}\ sa$ token into the $S.\texttt{CanConnect}\ sa\ skt$ token. The latter then allows the client to connect to the server, consuming the $S.\texttt{CanConnect}\ sa\ skt$ token to produce the channel endpoint ownership $c \succ^{\frac{sa.\texttt{ip}}{S.\texttt{cs}}} S.\texttt{prot}$. The channel endpoint ownership has the initial protocol state $S.\texttt{prot}$.

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 $c \succ^{\frac{ip}{ser}} prot$, 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 to receive a value, the protocol must be in a receiving state ($?\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\ prot$). If there is nothing to receive, we simply retain ownership of the original protocol state. Otherwise, we obtain 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, we derive the following specification for a blocking receive (`recv`) (which blocks until there is a value to return):

Ht-reliable-recv
$$\{c \succ^{\frac{ip}{ser}} ?\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\ prot\}\ \langle ip;\ \texttt{recv}\ c\rangle\ \{w.\ \exists\vec{y}.\ w = v[\vec{y}/\vec{x}] * c \succ^{\frac{ip}{ser}} prot[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]\}$$

2.4 A Simple Example: Verifying an Echo Server

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

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

[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 requests.

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 echo 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 5.2). 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 protocol from Section 2.2:

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

We thus start by instantiating the RCLib with the following user parameters:

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

Here the `S_ex.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_ex`) 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:

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

Until the session has been established, the proof of both the client and server is done by symbolic execution. Then, we can prove the server loops by Löb induction, by showing that at any given iteration, both loops end in the same state that they began. For the `accept_loop` this is straightforward, as the S.`Listens` skt token is preserved when applying Ht-accept [S]. For the `serve_loop`

[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.

DLM API:

$$\texttt{type dlm}$$

$$\texttt{val dlm\_start : saddr} \to \texttt{unit} \qquad \texttt{val dlm\_acquire : dlm} \to \texttt{unit}$$
$$\texttt{val dlm\_connect : saddr} \to \texttt{saddr} \to \texttt{dlm} \qquad \texttt{val dlm\_release : dlm} \to \texttt{unit}$$

DLM User Parameters and Resources:

$$UP \in \texttt{DLM\_UserParams} \triangleq \{\texttt{srv} : \texttt{Address};\quad R : \texttt{iProp}\}$$

$$S \in \texttt{DLM\_Resources}\ (UP : \texttt{DLM\_UserParams}) \triangleq \begin{Bmatrix} \texttt{CanStart} : \texttt{iProp}; & \texttt{CanAcquire} : \texttt{Ip} \to \texttt{Val} \to \texttt{iProp}; \\ \texttt{CanConnect} : \texttt{Address} \to \texttt{iProp}; & \texttt{CanRelease} : \texttt{Ip} \to \texttt{Val} \to \texttt{iProp} \end{Bmatrix}$$

DLM Specifications:

HT-DLM-START [S]
$\{S.\texttt{CanStart} * S.R\}$
$\langle S.\texttt{srv.ip}; \texttt{dlm\_start}\ S.\texttt{srv}\rangle$
$\{\texttt{True}\}$

HT-DLM-CONNECT [S]
$\{S.\texttt{CanConnect}\ sa\}$
$\langle sa.\texttt{ip}; \texttt{dlm\_connect}\ sa\ S.\texttt{srv}\rangle$
$\{dlm.\ S.\texttt{CanAcquire}\ sa.\texttt{ip}\ dlm\}$

HT-DLM-ACQUIRE [S]
$\{S.\texttt{CanAcquire}\ ip\ dlm\}$
$\langle ip; \texttt{dlm\_acquire}\ dlm\rangle$
$\{S.\texttt{CanRelease}\ ip\ dlm * S.R\}$

HT-DLM-RELEASE [S]
$\{S.\texttt{CanRelease}\ ip\ dlm * S.R\}$
$\langle ip; \texttt{dlm\_release}\ dlm\rangle$
$\{S.\texttt{CanAcquire}\ ip\ dlm\}$

Fig. 5. API and specifications for the Distributed Lock Manager

this is easy as well, as the `echo_prot` protocol recurses after two steps, so that the proof boils down to showing that the body of the loop adheres to the `echo_prot` protocol. This is straightforward to show, using HT-RELIABLE-RECV and HT-RELIABLE-SEND rules.

The verification of the client is a slightly more subtle, since the client sends two messages in a row, after which it awaits for two messages in a row, and as such this does not match syntactically with the `echo_prot`. However, it does so semantically, since the client's second send request and its first received response are independent, and so we can update the protocol by using the subprotocol relation, as we explained in Section 2.2. The propositions of the protocol ($m1 = |\text{“Carpe”}|$ and $m2 = |\text{“Diem”}|$) then lets us show that the assertions hold, which concludes the proof.

3 DISTRIBUTED LOCK MANAGER AND RPC SERVICE LIBRARIES

To demonstrate the expressivity of the reliable communication library of Section 2, we consider the specification and verification of two distributed components on top of our library, namely a *distributed lock manager* (Section 3.1), and a multi-threaded *RPC service* (Section 3.2), which we present in this section. In Section 4 we will then show how those libraries themselves are used to facilitate the formal development of clients and applications that make use of them.

3.1 Distributed Lock Manager

A distributed lock manager (DLM) enables coordination and coarse-grained concurrent synchronisation in a distributed network, for instance to enforce atomicity of transactions. Synchronisation is achieved by exposing a globally known *distributed lock*, which individual nodes can race to acquire before executing a critical section of the distributed algorithm (*e.g.*, a sequence of transactions).

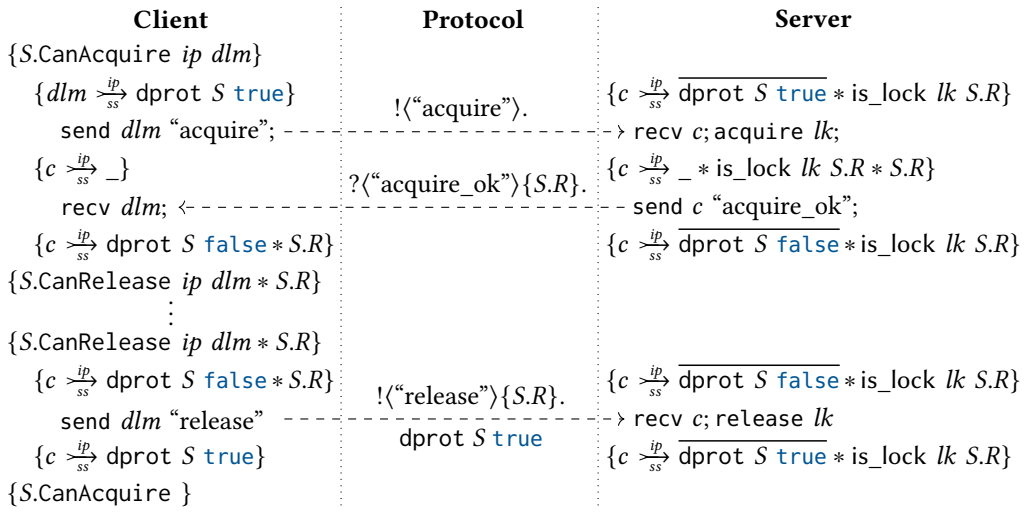

Fig. 6. The reliable communication of the DLM service

Figure 5 presents the API and the specification of the DLM. The server/client setup of the DLM is standard. The DLM service can be started on a server on any (globally known) socket address using `dlm_start`. Clients can then connect to the manager via a call to `dlm_connect`, which, when it terminates, returns an instance of the distributed lock object *dlm* (of the abstract data type `dlm`) which behaves locally as a standard lock: clients can call `dlm_acquire` to await and acquire the lock, and call `dlm_release` to release it.

3.1.1 Specifications of the DLM. The specifications of the DLM are stated using the same dependent specification pattern as the reliable communication library of Section 2. The specifications are parametric in the user provided parameters (*UP* : `DLM_UserParams`), consisting of a universally agreed upon manager network socket address (*S*.`srv`) and the resources (*S.R*) that the distributed lock will guard, once the manager is started. The DLM is then specified in terms of abstract resources (*S* : `DLM_Resources` *UP*) which reflect the state transition system of how a distributed lock can be created and used to synchronously access the guarded resources (*S.R*).

The initial resources *S*.`CanStart` and *S*.`CanConnect` *sa* (for each client) are freely obtained during the initial setup, similarly to the RCLib. The *S*.`CanStart` resource must be provided when starting the DLM service, to ensure that this only happens once, as specified by Ht-dlm-start [S]. The *S*.`CanConnect` *sa* resource is used once for each client to connect to the DLM service on the given socket address (*sa*), which in turn yields the *S*.`CanAcquire` *ip dlm* resource for the returned lock handle (*dlm*), as specified by Ht-dlm-connect [S]. The *S*.`CanAcquire` *ip dlm* resource is used when acquiring the lock, which in turn yields the guarded resources (*S.R*) and the *S*.`CanRelease` *ip dlm* resource, as specified by Ht-dlm-acquire [S]. Finally, the *S*.`CanRelease` *ip dlm* resource can be used, alongside the guarded resources (*S.R*), to release the lock, specified by Ht-dlm-release [S].

3.1.2 Verification of the DLM. The verification of the distributed lock manager primarily relies on the fact that the DLM service is implemented using a physical lock that is shared by the threads, one thread per client connected to the DLM, which serve the clients' "acquire" and "release" requests by trying to acquire/release the physical lock respectively. Consequently, all we need to do is verify this interplay of client-server communication and the physical lock, by employing our RCLib

specifications with the existing Aneris node-local lock specifications (omitted for brevity's sake). Similar to the example of the echo-server, to achieve this, we start by defining a suitable dependent separation protocol, which captures the client-side interaction between the clients and the DLM:

$$\texttt{dprot}\ (S : \texttt{DLM\_Resources}) \triangleq \mu rec.\, \lambda(b : \mathbb{B}).\, \texttt{if}\ b\ \texttt{then}\ \texttt{!}\langle\text{“acquire”}\rangle.\, \texttt{?}\langle\text{“acquire\_ok”}\rangle\{S.R\}.\, rec\ \neg b \ \texttt{else}\ \texttt{!}\langle\text{“release”}\rangle\{S.R\}.\, rec\ \neg b$$

The protocol can be in either of two states, as reflected by the boolean argument b: either the client can request the lock ($b = \texttt{true}$), by sending "acquire", or release the lock ($b = \texttt{false}$), by sending "release". Once an "acquire" message is sent, the server will reply with "acquire_ok" (when the physical lock becomes available), along with ownership of the guarded resources ($S.R$), after which the protocol goes to the release state (by flipping the boolean). To release the lock the client must send "release" along with the guarded resources ($S.R$), after which the protocol goes back to the acquire state (by flipping the boolean again).

Figure 6 sketches the proof of Ht-dlm-acquire [S] and Ht-dlm-release [S] to show the interplay between the reliable communication and the physical lock. In the proof of Ht-dlm-acquire [S] we start by unfolding the $S.\texttt{CanAcquire}\ sa.\texttt{ip}\ dlm$ abstract resource, which, under the hood, is simply the channel endpoint ownership ($dlm \overset{ip}{\underset{S.\texttt{srv.ip}}{\rightarrowtail}} \texttt{dprot}\ S\ \texttt{true}$). The protocol is then upheld by sending the "acquire" request to the DLM. When the DLM receives the request, it waits for and acquires the physical lock, obtaining the pre-established locked resources $S.R$. The DLM then sends back the "acquire_ok" message along with the pre-established resource $S.R$, in accordance with the protocol. Finally, the client receives the DLM response, along with the guarded resources. To obtain the post-condition of the Ht-dlm-acquire [S] specification, we wrap up the channel endpoint ownership ($dlm \overset{ip}{\underset{S.\texttt{srv.ip}}{\rightarrowtail}} \texttt{dprot}\ S\ \texttt{false}$), now in the release state, into its corresponding abstraction, which is exactly $S.\texttt{CanRelease}\ ip\ dlm$.

The verification of the Ht-dlm-release [S] specification is similar. The unfolded channel endpoint ownership, in its release state, is obtained from the release permission $S.\texttt{CanRelease}\ ip\ dlm$, and the guarded resources are sent back, according to the protocol. On the server side, the lock is simply released by giving back the received guarded resources. The post-condition of Ht-dlm-release [S] is then trivially satisfied by wrapping up the channel endpoint ownership, which has now returned to its acquire state, under the abstract resource $S.\texttt{CanAcquire}\ sa.\texttt{ip}\ dlm$.

3.2 RPC service

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 7 shows the API and the specifications of our RPC library. The RPC service 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

RPC API:

```
type ('a, 'b) rpc

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

RPC User Parameters and Resources:

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

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

RPC Specifications:

HT-RPC-CONNECT [S]

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

HT-RPC-START [S]

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

HT-RPC-REQUEST [S]

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

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

Fig. 7. Specifications for the RPC service

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.

3.2.1 Specifications of the RPC service. 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 are freely obtained during the initial setup, similar to the RCLib and DLM.

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 the incoming requests correctly. In particular, `rpc_process_spec` states that the procedure argument *qv* must satisfy the pre-established precondition S.`pre` *qv qd*, and that the results *rv* must satisfy the

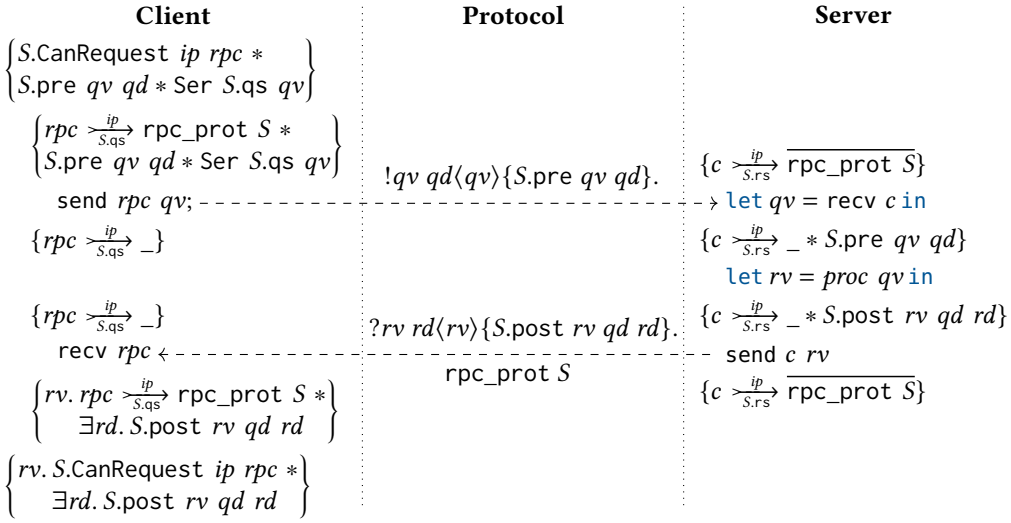

Fig. 8. The reliable communication of the RPC service

pre-established postcondition $S.\texttt{post}\ rv\ qd\ rd$. 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 $S.\texttt{CanConnect}\ sa$ resource is used once per client to connect to the RPC service on the given socket address, which in turn yields the $S.\texttt{CanRequest}\ ip\ rpc$ 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 $S.\texttt{CanRequest}\ ip\ rpc$ resource. Additionally, the argument qv must satisfy the pre-established precondition $S.\texttt{pre}\ qv\ qd$, and qv must be serializable by the pre-established request serializer $S.\texttt{qs}$. In return the client obtains the resources of the pre-established postcondition $S.\texttt{post}\ rv\ qd\ rd$ for the returned value rv.

3.2.2 Verification of the RPC service. The verification of the RPC service primarily boils down to showing that the specification of the client's `rpc_make_request` follows from the user provided proof of the specification `rpc_process_spec` of the request handler at the server side. The crux of this connection is once more to come up with an appropriate dependent separation protocol. In particular, we need to specify the delegation of the handler call to the server:

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

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

Figure 8 sketches the proof of how this protocol is used to connect the specifications of the client's local and remote calls in the verification of Ht-rpc-request [S]. First the abstract resource $S.\texttt{CanRequest}\ ip\ rpc$ is unfolded, to obtain the underlying channel endpoint ownership $rpc \overset{ip}{\underset{S.\texttt{qs}}{\rightarrowtail}} (\texttt{rpc\_prot}\ S)$. The RPC precondition resources for the request value ($S.\texttt{pre}\ qv\ qd$) are then transferred along the request, in accordance with the protocol. On the server side, the resources

are conversely received, in accordance with the protocol. The resources are then 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, again in accordance with the protocol. On the client side, the processed request and resources are finally received and returned. As the protocol completed one cycle of recursion and returned to the initial state, it can be 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, *e.g.*, replicated key-value store (KVS), 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 three important 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 will first consider a simplified version of the system, *i.e.*, a system consisting of only a single server: the leader. Considering only one server with no replication the system behaves as though the KVS is stored entirely locally — for the sake of this argument we are ignoring network issues such as delays or network partitions. Hence, we can give simple specifications to the read and write functions similar to those for local heap-allocated references:

LEADER-ONLY-WRITE-SPEC
$\{k \mapsto^{\mathsf{ldr}} vo\}\ \langle ip;\ write\ k\ v\rangle\ \{x.\ k \mapsto^{\mathsf{ldr}} \mathsf{Some}\ v * x = ()\}$

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 9). 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.

These are the important properties of observations[7]:

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

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

$$Obs(\mathrm{DB}, h) * Obs(\mathrm{DB}', h') \vdash \mathsf{prefix}\,(h, h') \vee \mathsf{prefix}\,(h', h) \qquad \text{(linear-order)}$$

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) states that all servers, the leader and the followers, always agree on the order of observed write events, *i.e.*, the history observed by one of them must be a prefix (as a sequence) of that of the other server.

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

[7] See our Coq development for a complete list of properties.

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
$$\left\{k \mapsto^{\text{kvs}}_{q} wo\right\} \langle ip;\ read\ k\rangle \left\{x.\ k \mapsto^{\text{kvs}}_{q} wo\ *\ ((x = \text{None} \wedge wo = \text{None}) \vee (\exists we.\ x = we.value \wedge wo = \text{Some } we))\right\}$$

FOLLOWER-READ-SPEC
$$\{Obs(\text{DB}_{fl}, h)\}\ read_{fl}\ k \left\{\begin{array}{r}x.\ \exists h'.\ \text{prefix}\,(h, h') * Obs(\text{DB}_{fl}, h') * \\ \left((x = \text{None} \wedge h'\big\downarrow_k = \text{None}) \vee (\exists we.\ h'\big\downarrow_k = \text{Some } we \wedge x = \text{Some } we)\right)\end{array}\right\}$$

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

The specification for reading a key from a follower, FOLLOWER-READ-SPEC, states that after reading we obtain the knowledge that the observed history on the follower in question is possibly extended in a way such that the returned write event is consistent with this observed history — the extended history is the history 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 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 simply obtain by unfolding the definition of $k \mapsto^{\text{ldr}} vo$ above, and a case analysis on whether vo is None or Some v.[8]

4.2 Client Examples

Here we discuss two illustrative examples of client programs. The examples are given in Figure 10. The example in Figure 10a demonstrates causal consistency of our KVS, which is implied by our specification as it guarantees linear histories. The example in Figure 10b shows how distributed locks can be used to make transactions. The code of the two examples are actually very similar. Both programs consist of two clients running in parallel. One client, `client0`, only performs two write operations, 37 to x followed by 1 to y, while the other client, `client1`, only reads. The first difference between the two examples is that in the causality example the read operation is directed at a follower while in the transaction example the read operation is directed at the leader. In the case of the causality example, `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*

[8]Technically, we also need the global invariant $\boxed{\text{GlobalInv}}^{\mathcal{N}}$ to derive the write specification, but we gloss over that here.

```
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 ()
```

(a) Causality Example

```
let do_writes lk =
  dlm_acquire lk;
  write "x" 37; write "y" 1;
  dlm_release lk

let rec do_reads lk =
    dlm_acquire lk;
    let vx = read "x" in
    if vx = Some 37 then
      let vy = read "y" in
      assert (vy = Some 1); dlm_release lk
    else
      dlm_release lk; do_reads lk


let client0 () = do_writes (dlm_connect ())

let client1 () = do_reads (dlm_connect ())

client0 () ||| client1 ()
```

(b) Transaction Example

Fig. 10. Two Examples Clients of Leader-Followers. In each case client0 and client1 are run 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.

depends on the first write. In the transaction example, on the other hand, `client1` waits until x has value 37, *i.e.*, it knows that the first write has been performed. It then proceeds to read the value of y and asserts that it must have value 1, *i.e.*, the second write is also performed. In other words, this example ensures that, at least as far as `client1` can observe, the two write operations form an atomic transaction; either they are both observable by `client1` or neither is. This only makes sense because both writes and reads are protected by a distributed lock which coordinates the two clients' interaction with the KVS.

We now sketch how we can prove that both of these client programs are safe, that is, that the assert statements shown in the two examples do not fail. (Formal specifications and proofs thereof can be found in our Coq formalization – our adequacy results described in section 5.2 then imply that both examples are safe to run and thus that the assert statements do not fail.)

Proof Intuition for the Causality Example. For this example we use an Iris invariant together with points-to propositions and the leader's observations (very 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. Hence, when `client1` observes the write to y it is guaranteed that the (only) write to x is also in the observed history.

Proof Intuition for the Transaction Example. The proof, instead of using an invariant as in the proof of the causality example above, takes advantage of the guarded resource of the distributed lock. The guarded resource essentially states the following simple property using points-to propositions: the value stored in x is 37 *if and only if* the value stored in y is 1. Hence, acquiring the lock on `client1` we know that if we read 1 from y then we must read 37 from x.

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.

Implementation. 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.[9]

Verification. The crux of the verification is to:

- Give concrete definitions of the abstract predicates, *e.g.*, $Obs(\mathrm{DB}, h)$ and $k \mapsto^{\mathrm{kvs}} wo$.
- 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
$KWTable\,(M)$	Tracks global view of the mapping from keys to their latest write events maintained by the leader.
$k \mapsto^{\mathrm{kvs}} wo$	As before; the write event always agrees with, *i.e.*, is included in, M in $KWTable\,(M)$.
$Log_{\mathrm{GI}}(\mathrm{DB}, h)$	Tracks the log of write events observed on server DB in the global invariant. Always agrees with Log_{LI} and Log_{local}.
$Log_{\mathrm{LI}}(\mathrm{DB}, h)$	Tracks the log of write events observed on server DB in the local invariant of the server. Always agrees with Log_{GI} and Log_{local}.
$Log_{\mathrm{local}}(\mathrm{DB}, h)$	Tracks the log of write events observed on server DB in the proof of correctness of RPC handlers. Always agrees with Log_{GI} and Log_{LI}.
$Obs(\mathrm{DB}, h)$	As before; the history h is a prefix of, *i.e.*, is included in, the history tracked in Log_{GI}, Log_{LI}, and Log_{local}.

[9] 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.

$$\textsc{Table-lookup}\quad KWTable\,(M) * k \mapsto^{\text{kvs}} wo \vdash M(k) = wo$$

$$\textsc{Table-update}\quad KWTable\,(M) * k \mapsto^{\text{kvs}} wo \Rrightarrow KWTable\,(M[k \mapsto wo']) * k \mapsto^{\text{kvs}} wo'$$

$$\textsc{Logs-agree}\quad \frac{X, Y \in \{\mathsf{GI}, \mathsf{LI}, \mathsf{local}\} \qquad X \neq Y}{Log_X(\mathsf{DB}, h) * Log_Y(\mathsf{DB}, h') \vdash h = h'}$$

$$\textsc{Obs-prefix}\quad \frac{X \in \{\mathsf{GI}, \mathsf{LI}, \mathsf{local}\}}{Log_X(\mathsf{DB}, h) * Obs(\mathsf{DB}, h') \vdash \mathsf{prefix}\,(h', h)}$$

$$\textsc{Obs-update}\quad \frac{\mathsf{prefix}\,(h, h')}{\begin{array}{c} Log_{\mathsf{GI}}(\mathsf{DB}, h) * Log_{\mathsf{LI}}(\mathsf{DB}, h) * Log_{\mathsf{local}}(\mathsf{DB}, h) \Rrightarrow \\ Log_{\mathsf{GI}}(\mathsf{DB}, h') * Log_{\mathsf{LI}}(\mathsf{DB}, h') * Log_{\mathsf{local}}(\mathsf{DB}, h') * Obs(\mathsf{DB}, h') \end{array}}$$

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

Here the propositions $KWTable\,(M)$ and $k \mapsto^{\text{kvs}} wo$ 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 $Log_{\mathsf{GI}}(\mathsf{DB}, h)$, $Log_{\mathsf{LI}}(\mathsf{DB}, h)$, and $Log_{\mathsf{local}}(\mathsf{DB}, h)$ are defined as the full part of an instance of the authoritative resource algebra (split into three different parts) while the proposition $Obs(\mathsf{DB}, h)$ is defined as a fragment in the same resource algebra.

The rules governing these propositions are shown in Figure 11. The rules capture how the inclusions of the underlying authoritative resource algebras are reflected for the propositions (notably in rules Table-lookup, Logs-agree, and Obs-prefix), and how they are preserved when resources are updated (notably in rules Table-update and Obs-update).
Given these propositions we can define the global and local invariants as follows:[10]

$$\mathsf{GlobalInv} \triangleq \exists M, h.\ KWTable\,(M) * Log_{\mathsf{GI}}(\mathsf{DB}_{ld}, h) * LogMapConsistent(h, M) * \ast_{fl \in Followers}\ \exists h'.\ Log_{\mathsf{GI}}(\mathsf{DB}_{fl}, h') * \mathsf{prefix}\,(h', h)$$

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

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 $\ell_{tbl_{\mathsf{DB}}}$. Similarly, it asserts that the server physically stores the sequence that is the history h, as the value v', in the memory location $\ell_{log_{\mathsf{DB}}}$.

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

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

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. The preconditions and postconditions of the read and write operations used in instantiating the RPC library are those of the more-general HOCAP-style specs.

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. Indeed, each layer in our stack of libraries (UDP networking primitives → reliable communication library → RPC library → KVS library) completely hides its internals, *i.e.*, its specs which the proof of its clients rely on, do not mention the previous layers in any way, shape, or form.

5 VERIFICATION OF THE RELIABLE COMMUNICATION LIBRARY AND ADEQUACY

We have thus far presented high-level specifications of the various reliable communication components that we have implemented and verified. In doing so, we have omitted some of the more technical details, for brevity's sake. In this section we address some of these omissions. In particular, we cover the verification of the reliable communication library (Section 5.1), and the process of obtaining the initial resources of our libraries, and thereby a closed proof in Aneris (Section 5.2).

5.1 Verification of the Reliable Communication Library

The crux of verifying the reliable communication library was to properly integrate it with the Actris framework. The Actris framework primarily operates on a notion of two reliable *logical buffers* between the two participants of the specified session, henceforth referred to as the *left* and the *right* participant. Each logical buffer tracks the values in transit from one participant to the other. we will refer to these as the *left-to-right* and *right-to-left* buffers, respectively. The buffers are reliable in that the order of messages are preserved, there are no duplicates, and messages are not lost. The dependent separation protocols of Actris then specify the values (and associated resources) that are allowed to pass through the logical buffers in either direction, depending on sequence and polarity (**!** or **?**) of the individual exchanges.

The main problem is then to connect these logical buffers to the messages that are exchanged over the unreliable network. As previously explained, the semantics of an unreliable network, like the one in Aneris, may cause messages to arrive out of order, be duplicated, or be lost. As a result, it may seem difficult to tie them to the reliable logical buffers of Actris, especially considering the dependent nature of Actris, which necessitates that messages are transferred in order.

To resolve this problem, we first establish a new intuition for the physical messages, by leveraging the implementation of the reliable transport layer, which is designed in the style of SCTP. In

particular, we will think of messages given to the transport layer as being *committed*, and messages received from the transport layer (in the order that they were sent) as being *accepted*.

With this intuition we can relate the committed and accepted messages with the logical Actris buffers. In particular, whenever either side commits a message, it is added to the corresponding buffer (e.g. the left-to-right buffer for the left participant, and vice versa) along with the associated resources, and when they accept messages, we remove it from the corresponding buffer (e.g. the left-to-right buffer for the right participant, and vice versa), and take out the associated resources.

Achieving an SCTP-like implementation is a matter of proving some handshake procedure for the connection establishment (which we do not present here due to the lack of space) and of employing standard reliability mechanisms such as sequence identifiers and retransmissions with acknowledgments. In particular, we ascribe any committed message with a sequence id. On the receiving side we then track the sequence id that we are currently expecting, and only accept messages with the right sequence id. With this we can guarantee that we only accept one message for each id (thus ensuring duplication-protection), and that we only accept the messages in order (order-preservation). To avoid the loss of messages, both sides additionally re-transmit all messages, until they receive an acknowledgement that the messages have been accepted (although we cannot formally verify that this guarantees progress as will be discussed in Section 7).

This approach relies on some subtle details. In particular, we need a way to:

(1) Atomically share and update the Actris logical buffers between the participants
(2) Guarantee that sent and received messages are properly tied to sequence ids
(3) Ensure that the sequenced messages correspond to the logical buffers of Actris

To achieve (1) we make use of Iris's invariants, which lets us share resources, which can then be accessed atomically. While this is a standard reasoning pattern in Iris, it proved non-trivial to use in conjunction with Actris. That is since the Actris rules require stripping multiple *later modalities* (as a result of Iris/Aneris being a step-indexed logic), while we canonically can only strip one, during an atomic step. To resolve this, we leveraged the work by Mével et al. [2019], that effectively allows the user to strip multiple laters per physical step (bounded by the total number of physical steps taken so far). We omit further details about step-indexing and stripping later modalities, and instead refer the interested reader to the related work [Hinrichsen et al. 2020; Mével et al. 2019].

To achieve (2) we use ghost state for multiple monotonically growing lists. In particular, we let each participant track the values that they have committed and accepted, and attach the duplicable *evidence* of any such commit and acceptance to the Aneris messages. This effectively guarantees that every committed message is associated with just one sequence id (one entry in the list of committed messages), and every message is only accepted once (one entry in the list of accepted messages, which is also a prefix of the other participants list of committed messages).

To achieve (3) we relate the logical buffers of Actris with the lists of committed and accepted messages. In particular, the left-to-right buffer is exactly the left participants list of committed messages, truncated by the right participants list of accepted messages, and vice versa.

5.2 Adequacy: obtaining the initial component resources and closed proofs

The component specification 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 a closed proof in Aneris is instantiated for a concrete network configuration, for which initial resources are provided and we can use those to derive the component-specific resources. This is formally captured by the foundationally mechanised Aneris adequacy theorem:

Theorem 5.1 (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:*

$$(\mathit{True} \Rrightarrow \exists \vec{x} : \vec{\tau}.\, P) \quad * \quad (\forall \vec{x} : \vec{\tau}.\, \exists \mathit{cfg}.\, \mathit{ip} \notin (\mathrm{dom}\ \mathit{cfg}) * \{P * \mathsf{NetRes}\ \mathit{cfg}\}\ \langle \mathit{ip};\ e \rangle\ \{\varphi\})$$

We then obtain the following properties:

- ***Safety:** The program e will never get stuck*
- ***Postcondition Validity:** If the program e terminates with value v, then φ v holds.*

In the remainder of this section, we outline how we apply this theorem. We focus on how the requisite resources necessary for each component are derived. This is somewhat technical and is intended for expert readers that are curious as to how we are able to obtain the initial component resources, and thereby complete and closed proofs.

To use the adequacy theorem, we first need to pre-allocate ghost state in the left-side proof obligation ($\mathsf{True} \Rrightarrow \exists \vec{x} : \vec{\tau}.\, P$), which the network configuration can then rely on. We then have to prove the right-side proof obligation where the universally quantified variables ($\vec{x} : \vec{\tau}$) lets us refer to the pre-allocated ghost names. The first step is then to pick the network configuration ($\mathit{cfg} : \mathsf{Ip} \xrightarrow{\text{fin}} \mathsf{Set}\ (\mathsf{Port} \times \mathsf{Option}\ (\mathsf{Val} \to \mathsf{iProp}))$), consisting of the network node ips (excluding the ip of the initial node), the open ports of the individual ips, and the statically known socket protocols. We must then prove the Hoare triple, in which we start with ownership of the pre-allocated ghost state P, and the initial network resources $\mathsf{NetRes}\ \mathit{cfg}$ (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. This is non-trivial, as the component-specific resources may depend on the socket interpretation protocols, which may depend on the specification user parameters, which may finally depend on the dynamically pre-allocated ghost state identifiers. This effectively means that we cannot derive the component-specific resources strictly after we have obtained $\mathsf{NetRes}\ \mathit{cfg}$.

To resolve this restriction, and allow the user to only be concerned about the user parameters, we provide a generic initialisation pattern for obtaining the initial component resources from the adequacy theorem. In particular, we need a way to let the user:

(1) Pre-allocate ghost state that the user parameters can depend on.
(2) Generate intermediate component-specific resources from the user parameters.
(3) Obtain the initial network resources including the component-specific ones.
(4) Extract the initial component-specific tokens from the initial network resources.

We achieve (1) directly from the left-side proof obligation of Theorem 5.1, where the user can freely pick the appropriate variables $\vec{x} : \vec{\tau}$ and proposition P, for the ghost state they need.
We achieve (2) from the following rule, which can be proven for each component:

$$\forall \mathit{UP}, \mathit{sas}.\, \mathsf{True} \Rrightarrow \exists S, \Psi.\, \mathtt{SrvPreRes}\ S\ \Psi * \left(\mathop{\Large *}_{\mathit{sa} \in \mathit{sas}} \mathtt{CltPreRes}\ S\ \Psi\ \mathit{sa} \right)$$

The rule states that given the component-specific user parameters (UP), and the set of client addresses (sas), we get the component-specific record (S), the component-specific server socket protocol (Ψ), and intermediate component-specific resources for the server and client ($\mathtt{SrvPreRes}\ S\ \Psi$ and $\mathop{\Large *}_{\mathit{sa} \in \mathit{sas}} \mathtt{CltPreRes}\ S\ \Psi\ \mathit{sa}$), which may contain component-specific ghost state.
We achieve (3) by picking the network configuration so that it includes the component-specific socket protocols in the right-side proof obligation of the adequacy theorem. This yields network resources that includes the component-specific socket protocol interpretations.

We achieve (4) from the following rule, which can be proven for each component:

$$\begin{array}{l}\mathsf{NetRes}\left(\mathit{cfg} \cup \begin{array}{l}\{[S.\mathtt{srv.ip} := \{(S.\mathtt{srv.port}, \mathsf{Some}\ \Psi)\}]\} \cup \\ \biguplus_{sa \in sas} \{[sa.\mathtt{ip} := \{(sa.\mathtt{port}, \mathsf{None})\}]\}\end{array}\right) * \\ \mathtt{SrvPreRes}\ S\ \Psi * (\divideontimes_{sa \in sas} \mathtt{CltPreRes}\ S\ \Psi\ sa) \Rrightarrow \\ \quad \mathsf{NetRes}\ \mathit{cfg} * S.\mathtt{SrvCanInit} * (\divideontimes_{sa \in sas} S.\mathtt{CltCanInit}\ sa) * \\ \quad \langle \text{component-specific specs} \rangle\end{array}$$

The rule captures how we can extract the component-specific server and client network resources and, in conjunction with the intermediate resources ($\mathtt{SrvPreRes}\ S\ \Psi$ and $\divideontimes_{sa \in sas} \mathtt{CltPreRes}\ S\ \Psi\ sa$), convert them into the initial server and client tokens ($S.\mathtt{SrvCanInit}$ and $\divideontimes_{sa \in sas} S.\mathtt{CltCanInit}\ sa$).

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 echo-server example:

```
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 root system node, we make use of the following Aneris rule:

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

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

6 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 has 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 transforer, 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 mechanism, where a default value of the expected type is returned, after which the sequence id is increased.

7 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 in 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. 2020. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. *CoRR* abs/2010.15030 (2020). arXiv:2010.15030 https://arxiv.org/abs/2010.15030

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

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

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*.

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