Verifying Reliable Sessions Over an Unreliable Network in Distributed Separation Logic

Léon Gondelman, Jonas Kastberg Hinrichsen, Mário Pereira, Amin Timany, Lars Birkedal

Cambium, Paris
January 16, 2023
Communicating processes

• Network communication & message-passing concurrency:
  
  —> coordination is done via exchanging messages (not via shared memory)

  —> communication protocols and ownership transfer play central role.

• One can expect that specification and reasoning about network and concurrency should exhibit common patterns and similar program logics.
Session types

**high-level typing pattern** to show safety for message-passing style concurrency

Example program:

```plaintext
let (c, c') := new_chan () in
fork {let x := recv c' in send c' (x + 2)};  // Service thread
send c 40; recv c                   // Client thread
```

Session types:

- $c : \text{chan (}!Z. ?Z. \text{end})$ and
- $c' : \text{chan (} ?Z. !Z. \text{end})$

Properties obtained:

- ✔ Program does not crash
- ✗ Program is correct (returns 42)
Actris Framework

**high-level specification pattern** to reason about for reliable message-passing communication [Hinrichsen et al. 2020]

- **Dependent Separation Protocols:**
  
  $$\text{prot} \in \text{iProto} ::= !\tilde{x} : \tilde{\tau} \langle v \rangle \{ P \}. \text{prot} \mid ?\tilde{x} : \tilde{\tau} \langle v \rangle \{ P \}. \text{prot} \mid \textbf{end}$$

  $$\text{echo_prot} \triangleq \mu \text{rec. } ?(s : \text{String}) \langle s \rangle . ! (n : \mathbb{N}) \langle n \rangle \{ n = |s| \} . \text{rec}$$

- **Specifications for message-passing concurrency:**

  \[
  \begin{align*}
  \{ c \mapsto !\tilde{x} : \tilde{\tau} \langle v \rangle \{ P \}. \text{prot} \ast P[\tilde{\tau}/\tilde{x}] \} \\
  \text{send } c \ (v[\tilde{t}/\tilde{x}]) \\
  \{ c \mapsto \text{prot}[\tilde{t}/\tilde{x}] \} \\
  \end{align*}
  \]

  \[
  \begin{align*}
  \{ c \mapsto ?\tilde{x} : \tilde{\tau} \langle v \rangle \{ P \}. \text{prot} \} \\
  \text{recv } c \\
  \{ w . \exists (\tilde{y} : \tilde{\tau}). (w = v[\tilde{y}/\tilde{x}]) \ast P[\tilde{y}/\tilde{x}] \ast c \mapsto \text{prot}[\tilde{y}/\tilde{x}] \} \\
  \end{align*}
  \]
Network Communication

Actris Session Type-based Reasoning

• provides a high-level model of reliable communication (Actris Ghost Theory)

• has been applied so far only to reason about message-passing concurrency, where the communication layer itself is reliable.

Network communication is fundamentally unreliable and asynchronous

• messages are lost, arrive out of order, got duplicated, or forged by adversary

• network partitions make it impossible to distinguish, in a finite amount of time, between delayed messages and lost messages (e.g. due to remote's crash)
How can we design a program logic for reliable network communication using session-typed based reasoning as high-level specification pattern?
Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems
Morten Krogh-Jespersen, Amin Timany, Marietta Oehlenschlager, Simon Oddershede Gregersen, and Lars Birkedal
Aarhus University, Aarhus, Denmark

Abstract. Building network-connected programs and distributed systems is a powerful way to provide scalability and availability in a digital, always-connected era. However, with great power comes great complexity. Reasoning about distributed systems is well-known to be difficult. In this paper we present Aneris, a novel framework based on separation logic that allows modular reasoning about distributed and concurrent systems. The framework supports higher-order store and network sockets, and is fully mechanized in the Coq proof assistant. We use our framework to verify an implementation of a load balancer that uses multi-threading to distribute load amongst multiple servers and an implementation of the two-phase-commit protocol with a replicated logging service as a client. The two examples certify that Aneris is well-suited for both horizontal and vertical modular reasoning.

Keywords: Distributed systems · Separation logic · Higher-order logic · Concurrency · Formal verification

1 Introduction
Reasoning about distributed systems is notoriously difficult due to their sheer complexity. This is partly due to the presence of malicious threats that can tamper with the state of distributed programs. In particular, as the number of nodes increases, the number of potential failure modes also increases exponentially. Under certain conditions, distributed computations have been made in the field of formal proofs of implementation of distributed protocols, such as the keychain protocol by Lamport [21] and the two-phase-commit protocol by Lamport [22]. However, in practice, developing such distributed systems can be challenging due to the complexity of the design, implementation, and verification process. Aneris is a novel framework based on separation logic that allows modular reasoning about distributed and concurrent systems.

AnerisLang, an OCaml-like language with

- UDP sockets primitives (msgs can be dropped, reordered or duplicated)
- Well-defined formal operational semantics
- Compiler from a subset of OCaml

Aneris Program Logic, a logic with

- All features from the Iris Framework (on top of which it is built in Coq)
- Proof rules to reason about node-local concurrency
- Proof rules to reason about UDP network communication

Hoare Logic ➔ Higher-Order Concurrent Separation Logic ➔ Distributed Separation Logic
Key contribution

We connect of the dependent session protocols of Actris to distributed systems, without extending the trusted code base of Aneris or Actris.

We achieve this

(1) by developing reliable communication library on top of Aneris' basic unreliable network primitives

(2) by proving the high-level Actris-like specifications of this library in Aneris, which involved coming up with a session escrow pattern
I. The API of the library
Our Library

- BSD sockets-like primitives
- 4-handshake connection
- buffered bidirectional channels
- sequence-ids/acknowledgments/retransmission mechanisms
- ~ 350 lines of OCaml

Some design choices:

- distinction between active/passive sockets and channels
- data transfer of serialisable values
Explicit distinction between active/passive socket and channel descriptor datatypes

open Ast

type ('a, 'b) client_skt
type ('a, 'b) server_skt
type ('a, 'b) chan_descr
val make_client_skt : 'a serializer -> 'b serializer -> saddr -> ('a, 'b) client_skt
val make_server_skt : 'a serializer -> 'b serializer -> saddr -> ('a, 'b) server_skt
val server_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
How **client** serialises values to be send to the **server**

```ocaml
open Ast

type ('a, 'b) client_skt
type ('a, 'b) server_skt
type ('a, 'b) chan descr
val make_client_skt : 'a serializer -> 'b serializer -> saddr -> ('a, 'b) client_skt
val make_server_skt : 'a serializer -> 'b serializer -> saddr -> ('a, 'b) server_skt
val server_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
```
OCaml API

How **server** serialises values to be send to the **client**

```ocaml
open Ast

type ('a, 'b) client_skt
type ('a, 'b) server_skt
type ('a, 'b) chan_descr
val make_client_skt : 'a serialzier -> 'b serialzier -> saddr -> ('a, 'b) client_skt
val make_server_skt : 'a serialzier -> 'b serialzier -> saddr -> ('a, 'b) server_skt
val server_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
```

How **client** deserialises values received from the **server**
Example: echo server

```ocaml
code

open Ast
open Serialization_code
open Client_server_code

let int_s = int_serializer
let str_s = string_serializer

let rec echo_loop c =
  let req = recv c in
  send c (strlen req);
  echo_loop c

let accept_loop s =
  let rec loop () =
    let c = fst (accept s) in
    fork echo_loop c; loop ()
  in loop ()

let server srv =
  let s = make_server_skf int_s str_s srv in
  server_listen s;
  fork accept_loop s

let client clt srv s1 s2 =
  let s = make_client_skf str_s int_s clt in
  let c = connect s srv in
  send c s1; send c s2;
  let m1 = recv c in
  let m2 = recv c in
  assert (m1 = strlen s1 && m2 = strlen s2)

let client_0 clt srv =
  client clt srv "carpe" "diem"
```

II. Specification
Spec 1/4 : Params & Resources

Our specification of the API primitives is dependent on

- the **user parameters** provided by the user

\[ UP \in RC\_UserParams \triangleq \{\text{srv} : \text{Address}; \ \text{prot} : \text{iProto}; \ \text{ss} : \text{Serializer}; \ \text{cs} : \text{Serializer}\} \]

- and the **abstract specification resources** provided by the library

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

**Notations** :  
\[ S := \text{SessionResources}(UP), \ S.\text{srv} := UP.\text{srv} \]
\textbf{Spec 2/4 : Client Setup}

\textbf{HT-MAKE-CLIENT-SOCKET} [S]
\{S.CltCanInit \textit{sa}\}
\langle \textit{sa}.ip; \text{mk}_\text{clt}_\text{skt} \; S.ss \; S.cs \; \textit{sa} \rangle
\{w. \exists \textit{skt}. \; w = \textit{skt} \ast S.\text{CanConnect} \; \textit{sa}.ip \; \textit{skt}\}

\textbf{HT-CONNECT} [S]
\{S.\text{CanConnect} \; \textit{ip} \; \textit{skt}\}
\langle \textit{ip}; \text{connect} \; \textit{skt} \; S.\text{srv} \rangle
\{w. \exists c. \; w = c \ast c \; \overset{\textit{sa}.ip}{\text{S.cs}} \; S.\text{prot}\}

channel endpoint ownership
**Spec 3/4 : Server Setup**

**HT-Make-Server-Socket [S]**
\[
\{S.SrvCanInit\}
\quad \langle S.srv.ip; \ mk_{-}srv_{-}skt \ S.ss \ S.cs \ S.srv \rangle
\quad \{w. \ \exists skt. \ w = skt * S.CanListen \ skt \}
\]

**HT-Listen [S]**
\[
\{S.CanListen \ skt\}
\quad \langle S.srv.ip; \ listen \ skt \rangle
\quad \{S.Listens \ skt\}
\]

**HT-Accept [S]**
\[
\{S.Listens \ skt\}
\quad \langle S.srv.ip; \ accept \ skt \rangle
\quad \{w. \ \exists c, sa. \ w = (c, sa) * S.Listens \ skt * c \rightarrow \ \frac{S.srv.ip}{S.ss} \rightarrow S.prot\}
\]

---

channel endpoint ownership
Spec 4/4: Send and Receive

**HT-reliable-send**

\[
\{ c \xrightarrow{\text{ip}}_{\text{ser}} \! \vec{x} : \vec{\tau} \langle v \rangle \{ P \}. \text{prot} \ast P[\vec{t}/\vec{x}] \ast \text{Ser} \ \text{ser} \ (v[\vec{t}/\vec{x}]) \} \\
\langle \text{ip}; \ \text{send} \ c \ (v[\vec{t}/\vec{x}]) \rangle
\]

\[
\{ c \xrightarrow{\text{ip}}_{\text{ser}} \text{prot}[\vec{t}/\vec{x}] \}
\]

**HT-reliable-receive**

\[
\{ c \xrightarrow{\text{ip}}_{\text{ser}} ?\vec{x} : \vec{\tau} \langle v \rangle \{ P \}. \text{prot} \} \\
\langle \text{ip}; \ \text{recv} \ c \rangle
\]

\[
\{ w. \ \exists \vec{y}. \ w = v[\vec{y}/\vec{x}] \ast c \xrightarrow{\text{ip}}_{\text{ser}} \text{prot}[\vec{y}/\vec{x}] \ast P[\vec{y}/\vec{x}] \}
\]

These specs are similar to the Actris specs for message-passing concurrency and they are the same for both channel endpoints.
- (Step 1) Writing the program(s) in the OCaml subset (done by user)
- (Step 2) Translating the programs to AnerisLang (done by compiler)
- (Step 3) Defining a Dependent Separation Protocol (done by user)
- (Step 4) Verifying each node individually (done by user)
- (Step 5) Applying the adequacy theorem to obtain a closed proof, i.e., a proof in Coq independent of Iris and Aneris, (done by user).
Step 1: Write OCaml sources.

```ocaml
let rec echo_loop c =
  let rec req = recv c in
  send c (strlen req);
  echo_loop c
```

Definition `echo_loop`:

```
Definition echo_loop : val :=
  rec: "echo_loop" "c" :=
  let: "req" := recv "c" in
  send "c" (strlen "req");
  "echo_loop" "c".
```

Step 2: Generate Coq definition

Step 3: Define the dependent separation protocol.

Definition `prot_aux`:

```
Definition prot_aux (rec : iProto Σ) : iProto Σ :=
  (<! (s : string)> MSG #s ;
  <! (n : N) > MSG #n {{ rString.length s = n }}; rec)%proto.
```
Step 4. Instantiate the following class for echo server...

```
Class Reliable_communication_init := {
    Reliable_communication_init_setup
    E (UP : Reliable_communication_service_params):
    ~RCPParams_srv_N \leq E \rightarrow
    \exists \text{ Chan_mapsto_resource),}
    \exists \text{ SessionResources UP),}
    \text{ SrvInit, }
    \text{ Client_skt_spec UP SnRes, }
    \text{ Server_skt_spec UP SnRes, }
    \text{ connect spec UP SnRes, }
    \text{ server_listen spec UP SnRes, }
    \text{ accept spec UP SnRes, }
    \text{ send spec, }
    \text{ send_spec_tele, }
    \text{ try_recv spec, }
    \text{ recv spec}
}.
```
...and verify each node separately (modular proof).

Lemma wp_echo_loop c :
  {{} { c ⊣{S.srv_saddr_ip, S.srv_ser} iProto_dual S.protocol }}
  echo_loop c @[S.srv_saddr_ip]
  {{} v, RET v ; ⊥ }}
Proof.
iIntros (Φ) "Hci Φ". iLöb as "IH". wp_lam.
wp_recv (s₁) as "_". wp_send with "[///]".
wp_seq.by iApply ("IH" with "[$Hci]").
Qed.

Step 5. Apply the adequacy theorem to obtain a closed proof, i.e.,
a proof in Coq independent of Iris and Aneris.
Case study: Remote Procedure Call

So far:

- **from** Aneris rules to reason about UDP
- **to** the logical rules for Client-Server Sessions

Distributed components:

- **from** rules for Client Server Sessions
- **to** the Remote Procedure Call (RPC) library

The RPC abstraction specification allows to reason about distributed applications (e.g. key-value store) **without any reasoning about network-level communication at all.**
The API exposes just one service handler, but in which the types of request and response are polymorphic and higher-order.

- instantiating those types with sum-types $\tau_r^1 + \tau_r^2$ (for requests), and $\tau_r^1 + \tau_r^2$ (for responses) allows us to encode an RPC service that handles multiple procedures calls e.g., as a pair of procedures of type $\tau_q^1 \rightarrow \tau_r^1$ and $\tau_q^2 \rightarrow \tau_r^2$. 

```haskell
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
```
As before, we use the **dependent specification pattern**, starting with user’s parameters and library’s abstract resources:

**RPC User Parameters and Resources:**

\[
UP \in \text{RPC\_UserParams} \triangleq \\
\{ \text{srv} : \text{Address}; \quad \text{ReqData} : \text{Type}; \quad \text{RepData} : \text{Type}; \\
\quad \text{qs} : \text{Serializer}; \quad \pre : \text{Val} \to \text{ReqData} \to \text{iProp}; \\
\quad \text{rs} : \text{Serializer}; \quad \post : \text{Val} \to \text{ReqData} \to \text{RepData} \to \text{iProp} \}
\]

\[
S \in \text{RPC\_Resources} \,(UP : \text{RPC\_UserParams}) \triangleq \\
\{ \text{CanStart} : \text{iProp}; \quad \text{CanConnect} : \text{Address} \to \text{iProp}; \quad \text{CanRequest} : \text{Ip} \to \text{Val} \to \text{iProp} \}
\]
<table>
<thead>
<tr>
<th>Client-side</th>
<th>Server-side</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HT-rpc-connect [S]</strong></td>
<td><strong>rpc_process_spec S proc (= \forall qv, qd.</strong></td>
</tr>
<tr>
<td>({S.\text{CanConnect } sa})</td>
<td>({S.\text{pre } qv qd})</td>
</tr>
<tr>
<td>(\langle sa.ip; \text{rpc_connect } S.qs S.rs sa S.srv\rangle)</td>
<td>(\langle S.srv.ip; \text{proc } qv\rangle)</td>
</tr>
<tr>
<td>({\text{rpc. } S.\text{CanRequest } sa.ip rpc})</td>
<td>({rv. \exists rd. \text{Ser } S.rs rv \ast S.\text{post } rv qd rd})</td>
</tr>
</tbody>
</table>

| **HT-rpc-request [S]**                          | **HT-rpc-start [S]**                                                      |
| \(\{S.\text{CanRequest } ip rpc \ast \}\)      | **rpc_process_spec S proc**                                              |
| \(\{S.\text{pre } qv qd \ast \text{Ser } S.qs qv\}\) | \(\{S.srv.ip; \text{rpc\_start } S.rs S.qs S.srv proc\}\)              |
| \(\langle ip; \text{rpc\_make\_request } rpc qv\rangle\) | \(\{\text{True}\}\)                                                   |
| \(\{rv. S.\text{CanRequest } ip rpc \ast \exists rd. S.\text{post } rv qd rd\}\) |                                                            |
HT-rpc-connect [S]
\{ S.CanConnect sa \}
\langle sa.ip; rpc_connect S.qs S.rs sa S.srv \rangle
\{ rpc.S.CanRequest sa.ip rpc \}

HT-rpc-request [S]
\{ S.CanRequest ip rpc * \}
\{ S.pre qv qd * Ser S.qs qv \}
\langle ip; rpc_make_request rpc qv \rangle
\{ rv. S.CanRequest ip rpc * \\exists rd. S.post rv qd rd \}

HT-rpc-process spec S proc \triangleq \forall qv, qd.
\{ S.pre qv qd \}
\langle S.srv.ip; proc qv \rangle
\{ rv. \exists rd. Ser S.rs rv * S.post rv qd rd \}

HT-rpc-start [S]
\{ S.CanStart * rpc_process_spec S proc \}
\langle S.srv.ip; rpc_start S.rs S.qs S.srv proc \rangle
\{ True \}
let service_loop c (request_handler : 'req -> 'rep) () : unit =
  let rec loop () =
    let req = recv c in
    let rep = request_handler req in
    send c rep; loop ()
in loop ()

let accept_new_connections_loop skt request_handler () : unit =
  let rec loop () =
    let new_conn = accept skt in
    let (c, _d) = new_conn in
    fork (service_loop c request_handler) (); loop ()
in loop ()

let run_server
  (ser[@metavar] : 'repl serializer) (deser[@metavar] : 'req serializer) addr
  (request_handler : 'req -> 'rep) : unit =
  let (skt : ('repl, 'req) server_skt) = make_server_skt ser deser addr in
  server_listen skt;
  fork (accept_new_connections_loop skt request_handler) ()
Dependent Separation Protocol:

\[
\text{rpc_prot} (S : \text{RPC\_Resources UP}) \triangleq \\
\mu\text{rec.}! (q_v : \text{Val}) (q_d : S.\text{ReqData}) \langle q_v \rangle \{ S.\text{pre} q_v q_d \}. \\
\nu (r_v : \text{Val}) (r_d : S.\text{RepData}) \langle r_v \rangle \{ S.\text{post} r_v q_d r_d \}. \text{rec}
\]
### RPC Verification (3/3)

**Client**

\[
\begin{align*}
\{ & S.\text{CanRequest} \ ip \ rpc \ * \\
& \{ S.\text{pre} \ qv \ qd \ * \ \text{Ser} \ S.\text{qs} \ qv \} \\
\{ & rpc \ \xrightarrow{ip} \ _{s,qs} \ \text{rpc} \_prot \ S \ * \\
& \{ S.\text{pre} \ qv \ qd \ * \ \text{Ser} \ S.\text{qs} \ qv \} \\
\text{send} & \ \text{rpc} \ \text{qv}; \\
\{ & rpc \ \xrightarrow{ip} \ _{s,qs} \ _{\text{eq}} \\
\text{recv} & \ \text{rpc} \ \xrightarrow{ip} \ _{s,qs} \\
\{ & rv. \ \text{rpc} \ \xrightarrow{ip} \ _{s,qs} \ \text{rpc} \_prot \ S \ * \\
& \quad \exists rd. \ S.\text{post} \ rv \ qd \ rd \} \\
\{ & rv. \ S.\text{CanRequest} \ ip \ rpc \ * \\
& \quad \exists rd. \ S.\text{post} \ rv \ qd \ rd \}
\end{align*}
\]

**Protocol**

\[
\begin{align*}
& qv \ qd \ (qv) \ \{ S.\text{pre} \ qv \ qd \}.
\end{align*}
\]

**Server**

\[
\begin{align*}
\{ & c \ \xrightarrow{ip} \ _{s,rs} \ \text{rpc} \_prot \ S \} \\
& \quad \text{let} \ qv = \ \text{recv} \ c \ \text{in} \\
\{ & c \ \xrightarrow{ip} \ _{s,rs} \ _{\text{eq}} \ * \ S.\text{pre} \ qv \ qd \} \\
& \quad \text{let} \ \text{rv} = \ \text{proc} \ qv \ \text{in} \\
\{ & c \ \xrightarrow{ip} \ _{s,rs} \ _{\text{eq}} \ * \ S.\text{post} \ rv \ qd \ rd \} \\
& \quad \text{send} \ c \ \text{rv} \\
\{ & c \ \xrightarrow{ip} \ _{s,rs} \ \text{rpc} \_prot \ S \}
\end{align*}
\]
Modular reasoning about distributed applications

LEADER-ONLY-READ-SPEC
\[ \{ k \rightleftharpoons_{q}^{ldr} vo \langle ip; read k \rangle \{ x. k \rightleftharpoons_{q}^{ldr} vo * x = vo \} \]

LEADER-ONLY-WRITE-SPEC
\[ \{ k \rightleftharpoons_{q}^{ldr} vo \langle ip; write k v \rangle \{ x. k \rightleftharpoons_{q}^{ldr} \text{Some } v * x = () \} \]

(Distributed Key-Value Store with Leader-Followers)
III. Verification
Verification (of established sessions)

To understand what is the **crux of the verification** (for the code when session is established), we need to take a look on

1. how **resources are transferred** for unreliable communication in Aneris Logic
2. how the **reliable transfer is modelled** in Actris Ghost Theory

The proof then proceeds in two steps:

1. connecting Actris Ghost Theory & Aneris Logic  (**Session Escrow Pattern**)
2. verifying the implementation (**API send/receive and internal procedures**
Resource Transfer in Aneris

In Aneris, safe transfer of spatial resources (associated with a sent message) over the unreliable network is achieved by

- storing the spatial resources in a shared logical context (Iris invariant),
- and then sending a duplicable witness over the network

This (escrow pattern) enables retransmission (as the witness is duplicable), and safe transfer (as the spatial resources can only be taken out once).

However, it does not allow dependencies between the resources stored in the shared logical context (indeed, there might be several resources in transit).
Reliable transfer is modelled using logical buffers $\vec{v}_1, \vec{v}_2$ which

- describe **symmetrically** for each direction the messages in transit
- are governed (inside an Iris invariant) by the shared resource $\text{prot}\_\text{ctx} \; \chi \; \vec{v}_1 \; \vec{v}_2$

\[
\text{True} \Rightarrow \exists \chi. \; \text{prot}\_\text{ctx} \; \chi \; {\epsilon} * \; \text{prot}\_\text{ownl} \; \chi \; \text{prot} * \; \text{prot}\_\text{ownr} \; \chi \; \text{prot}
\]

\[
\text{prot}\_\text{ctx} \; \chi \; \vec{v}_1 \; \vec{v}_2 \; * \; \text{prot}\_\text{ownl} \; \chi \; \left( ! \vec{x} : \vec{t} \langle v \rangle \{ P \} . \; \text{prot} \right) * \; P[\vec{t}/\vec{x}] \Rightarrow
\]

\[
\left( \Rightarrow \vec{v}_2 \rightarrow \text{prot}\_\text{ctx} \; \chi \; (\vec{v}_1 \; [v[\vec{t}/\vec{x}]] \; \vec{v}_2) \; * \; \text{prot}\_\text{ownl} \; \chi \; (\text{prot}[\vec{t}/\vec{x}]) \right)
\]

\[
\text{prot}\_\text{ctx} \; \chi \; \vec{v}_1 \; (\; [w] \; \cdot \; \vec{v}_2) \; * \; \text{prot}\_\text{ownl} \; \chi \; (\; ?\vec{x} : \vec{t} \langle v \rangle \{ P \} . \; \text{prot} \;) \Rightarrow
\]

\[
\Rightarrow \exists \vec{y}. \; (w = v[\vec{y}/\vec{x}]) \; * \; P[\vec{y}/\vec{x}] \; * \; \text{prot}\_\text{ctx} \; \chi \; \vec{v}_1 \; \vec{v}_2 \; * \; \text{prot}\_\text{ownl} \; \chi \; \text{prot}[\vec{y}/\vec{x}] 
\]
Actris Ghost Theory allows dependencies between the resources stored in the shared logical context.

However,

- as such it does not use an escrow pattern, which is needed to connect Actris logical state with the spatial transfer using duplicable witnesses.

- the duplicable witnesses must appropriately reflect the Actris logical state so that resources can be acquired in accordance to their dependence.
Message Histories

• We introduce **additional logical buffers** $T_l$, $R_l$, $T_r$, $R_r$ as a glue.

  $(T_l, T_r)$ describe the **history of sent** messages;
  $(R_l, R_r)$ describe the **history of received** messages (by the application).

• Various **relations** hold between Actris, glue, and physical buffers:

  • Rr is prefix of Tl and Rl is prefix of Tr \hspace{1cm} (Internal-Coh)
  • $v_1 = T_l - R_r$ and $v_2 = T_r - R_l$ \hspace{1cm} (Actris-Coh)
  • sbufl is suffix of Tl and sbufr is suffix of Tr \hspace{1cm} (SBuf-Coh)
  • rbufl is prefix of $(T_r - R_l)$ and rbufr is prefix of $(T_l - R_r)$ \hspace{1cm} (Rbuf-Coh)
Session Escrow Pattern

The monotonic list ghost theory:

$$\textbf{AUTH-LIST-ALLOC}$$

$$\text{True} \Rightarrow \exists \gamma. \text{auth} \_\text{list} \gamma \in \ast \text{ list} \_\text{len} \gamma \ 0$$

$$\textbf{AUTH-LIST-EXTEND}$$

$$\text{auth} \_\text{list} \gamma \vec{x} \ast \text{list} \_\text{len} \gamma \ n \Rightarrow \text{auth} \_\text{list} \gamma \ (x \cdot [\vec{x}]) \ast \text{list} \_\text{len} \gamma \ (n + 1) \ast \text{frag} \_\text{list} \gamma \ n \ x$$

$$\textbf{AUTH-LIST-AGREE}$$

$$\text{auth} \_\text{list} \gamma \vec{x} \text{ frag} \_\text{list} \gamma \ i \ x \hspace{2cm} \frac{}{\vec{x}_i = x}$$

$$\textbf{AUTH-LIST-LENGTH}$$

$$\text{auth} \_\text{list} \gamma \vec{x} \ \text{list} \_\text{len} \gamma \ n \hspace{2cm} \frac{}{|\vec{x}| = n}$$

$$\textbf{FRAG-LIST-DUP}$$

$$\text{frag} \_\text{list} \gamma \ i \ x \ast \text{frag} \_\text{list} \gamma \ i \ x$$

Shared logical context (Iris invariant):

$$\exists Tl, Tr, Rl, Rr. \text{auth} \_\text{list} \chi_{T_1} Tl * \text{auth} \_\text{list} \chi_{T_r} Tr * \text{auth} \_\text{list} \chi_{R_1} Rl * \text{auth} \_\text{list} \chi_{R_r} Rr *$$

$$\text{prot} \_\text{ctx} \chi_{\text{chan}} (Tl - Rr) (Tr - Rl) * Rr \leq_p Tl * Rl \leq_p Tr * \vec{x} |Tl| * \vec{x} |Tr|$$

Duplicable witnesses: $$\text{frag} \_\text{list} \chi_{T_1} n \ v, \text{frag} \_\text{list} \chi_{T_r} i \ v$$
// Session, omitting fragments about right side
session γTl γTlc γRr ≜ ∃ Tl n, prot_ctx (drop n Tl) _ * auth_list γTl Tl * auth_count γTlc |Tl| * auth_count γRr n

// Session Escrow Rule for Send
session γTl γTlc γRr ⊢ prot_own_l (! xs <v> { Q } . p) * frag_count γTlc n * Q ==> prot_own_l p * frag_count γTlc (S n) * frag_list γTl n v

// Session Escrow Rule forRecv
session γTl γTlc γRr ⊢ prot_own_r (? xs <v> { Q } . p) * frag_count γRr n * frag_list γTl n v ==> prot_own_r p * frag_count γRr (S n) * Q
The internal procedures that enforce the fault-tolerance are also (mostly) the same for clients and servers, and so are our proofs.
Other Observations (2/3)

- The 4-handshake is different for each side and requires some effort in verification as it encodes an STS with several edge and absurd cases.
The implementation/verification of server side is more difficult, because the server must maintain a **table of known clients with their connection state** and a **channel description queue** for the established connections.
V. Conclusion & Future Directions
Possible Future Directions

- **Graceful/Abrupt session ending**: detectable connection failures, reconnection
- **Cryptography/Security**: 4-way handshake procedure / authentification / QUIC
- **Network Partitions**: group membership/consensus built on top of our library
- **Group Communication**: client-service communication
- **Transparency**: verified libs for distributed/multithreaded programs (e.g. Functory)
- (and maybe your insights/ideas !)
Thank you!
Backup slides
Remark: the proof rules for UDP primitives are low-level, but what we need is to achieve **expressive specifications** that abstract away most of low-level details!
### POSSIBLE SOLUTIONS

<table>
<thead>
<tr>
<th>POSSIBLE SOLUTION</th>
<th>general-purpose solution</th>
<th>trusted code base</th>
<th>high-level specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>implement and verify reliability ad hoc for each application</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>extend Aneris semantics and logics with reliable sessions primitives</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>implement and verify a transport layer library on top of UDP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Aneris Distributed Separation Logic

**HT-send**
\[
\begin{aligned}
\{sh \xrightarrow{m\text{.src\_ip}} (\text{Some}(m\text{.src}), b) * m\text{.dst} \Rightarrow \Phi * \\
m\text{.src} \Rightarrow (R, T) * (m \notin T \Rightarrow \Phi m) \\
\langle m\text{.src\_ip}; \text{sendto} sh m\text{.str} m\text{.dst} \rangle \\
w. w = |m\text{.src}| * m\text{.src} \Rightarrow (R, T \cup \{m\}) * \\
sh \xrightarrow{m\text{.src\_ip}} (\text{Some}(m\text{.src}), b)
\end{aligned}
\]

**HT-reCV**
\[
\begin{aligned}
\{sh \xrightarrow{sa\_ip} (\text{Some}(sa), b) * sa \Rightarrow (R, T) * sa \Rightarrow \Phi \}
\langle sa\_ip; \text{receivefrom} sh \rangle \\
w. sh \xrightarrow{sa\_ip} (\text{Some}(sa), b) * \\
(b = \text{false} * w = \text{None} * sa \Rightarrow (R, T)) \lor \\
(\exists m. w = \text{Some}(m\text{.str}, m\text{.src}) * m\text{.dst} = sa * \\
\text{sa} \Rightarrow (R \cup \{m\}, T) * (m \notin R \Rightarrow \Phi m))
\end{aligned}
\]

(a) socket handle resource \( sh \xrightarrow{sa\_ip} (\text{Some}(sa), b) \)
(b) message history resources  \( sa \sim (R, T) \)
Aneris Distributed Separation Logic

\[
\begin{align*}
\text{HT-SEND} & \quad \{ sh \xrightarrow{m.\text{src}_\text{ip}} \text{Some}(m.\text{src}), b) \times m.\text{dst} \Rightarrow \Phi \times \} \\
& \quad \{ m.\text{src} \xrightarrow{(R, T)} \times (m \notin T \Rightarrow \Phi \times m) \} \\
& \quad \langle m.\text{src}_\text{ip}; \text{sendto} \, sh \, m.\text{str} \, m.\text{dst} \rangle \\
& \quad \{ w. \, w = |m.\text{src}| \times m.\text{src} \xrightarrow{(R, T \cup \{m\})} \times \} \\
& \quad \{ sh \xrightarrow{m.\text{src}_\text{ip}} \text{Some}(m.\text{src}), b) \} \\
\end{align*}
\]

\[
\begin{align*}
\text{HT-RECV} & \quad \{ sh \xrightarrow{sa_{\text{ip}}} \text{Some}(sa), b) \times sa \xrightarrow{(R, T)} \times sa \Rightarrow \Phi \} \\
& \quad \langle sa_{\text{ip}}; \text{receivefrom} \, sh \rangle \\
& \quad \{ w. \, sh \xrightarrow{sa_{\text{ip}}} \text{Some}(sa), b) \times \} \\
& \quad \{ (b = \text{false} \times w = \text{None} \times sa \xrightarrow{(R, T)}) \lor \} \\
& \quad \{ (\exists m. \, w = \text{Some} (m.\text{str}, m.\text{src}) \times m.\text{dst} = sa \times \} \\
& \quad \{ sa \xrightarrow{(R \cup \{m\}, T)} \times (m \notin R \Rightarrow \Phi \times m) \} \\
\end{align*}
\]

\( (c) \) socket protocol predicate \( sa \Rightarrow \Phi \)