Verifying Reliable Sessions Over an Unreliable Network in Distributed Separation Logic

Léon Gondelman & Jonas Kastberg Hinrichsen

The Second Iris Workshop
May 2, 2022
I. Reliable Communication in Distributed Systems
Communicating processes

- Network communication & message-passing concurrency:

  > coordination is done via **exchanging messages** (not via shared memory)

  > **communication protocols** and **resource transfer** play central role
Fundamental Difference

Communication over the network is fundamentally **unreliable** and **asynchronous**:

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

> messages arrive from one machine to another with a certain **delay**

> **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)
Fault Tolerance

- **Transport layer protocols** such as TCP, SCTP and others provide some reliability guarantees (*at-most-once in-order delivery*).

- However, no protocol can guarantee that messages *will arrive in-order & without duplicates exactly once*.

- In the presence of network partitions/broken connections, TCP is no better than UDP: *in fine*, reliability is achieved at the application level.

- Many reasons to build fault-tolerance on top of UDP: > gaming community, Google QUIC (2013), Ensemble (Haiden 98)
Verification Perspective

- Two research directions:
  - **Assume** fault-tolerance **to reason** about high-level problems/algorithms: *map-reduce, deadlock freedom, op-based CRDTs, …*
  - **Model** network with faults **to build** fault-tolerance: *consensus algorithms, reliable causal broadcast, client-server sessions.*

- **Longstanding goal:** a unified framework where high-level abstractions meet realistic fault-tolerant implementations.

- **The story of this work:** one step towards this goal.
Key Observation (1/2)

- **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.

\[
\begin{align*}
\{ c \mapsto !\bar{x} : \bar{\tau} \langle v \rangle \{ P \}. \text{prot} \} + P[\bar{t} / \bar{x}] & \quad \{ c \mapsto ?\bar{x} : \bar{\tau} \langle v \rangle \{ P \}. \text{prot} \} \\text{send} c (v[\bar{t} / \bar{x}]) \\
\{ c \mapsto \text{prot}[\bar{t} / \bar{x}] \} & \quad \{ w. \exists (\bar{y} : \bar{\tau}). (w = v[\bar{y} / \bar{x}]) * \\
& \quad \quad P[\bar{y} / \bar{x}] * c \mapsto \text{prot}[\bar{y} / \bar{x}] \} 
\end{align*}
\]
Aneris Distributed Separation Logic

> provides rules to reason about unreliable unconnected communication;

> had no native/library support for reliable/connected communication
(i.e. each time reliability/sessions had to be built in ad-hoc way).

(a) socket handle resource  \( sh \xrightarrow{\text{sa}_\text{ip}} \text{(Some}(sa), b) \)
• **Aneris Distributed Separation Logic**

  > provides rules to reason about unreliable unconnected communication;

  > had no native/library support for reliable/connected communication (i.e. each time reliability/sessions had to be built in ad-hoc way).

\[
\text{HT-SEND}
\begin{align*}
& \{ sh \xrightarrow{m.\text{src}} (\text{Some}(m.\text{src}), b) \ast m.\text{dst} \Rightarrow \Phi \ast \\
& \quad m.\text{src} \Rightarrow (R, T) \ast (m \not\in T \Rightarrow \Phi m) \\
& \quad \langle m.\text{src} \_i; \text{sendto } sh \text{ m.str m.dst} \rangle \\
& \quad \{ w. w = |m.\text{src}| \ast m.\text{src} \Rightarrow (R, T \cup \{m\}) \ast \\
& \quad \quad sh \xrightarrow{m.\text{src}} (\text{Some}(m.\text{src}), b) \}
\end{align*}
\]

\[
\text{HT-RECV}
\begin{align*}
& \{ sh \xrightarrow{\text{sa}_{\text{ip}}} (\text{Some}(sa), b) \ast sa \Rightarrow (R, T) \ast sa \Rightarrow \Phi \}
\\
& \langle \text{sa}_{\text{ip}}; \text{receivefrom } sh \rangle \\
& \{ w. sh \xrightarrow{\text{sa}_{\text{ip}}} (\text{Some}(sa), b) \ast \\
& \quad (b = \text{false} \ast w = \text{None} \ast sa \Rightarrow (R, T)) \lor \\
& \quad (\exists m. w = \text{Some}(m.\text{str}, m.\text{src}) \ast m.\text{dst} = sa \ast \\
& \quad \quad sa \Rightarrow (R \cup \{m\}, T) \ast (m \not\in R \Rightarrow \Phi m)) \}
\end{align*}
\]

\( (b) \text{ message history resources } \quad sa \Rightarrow (R, T) \)
Key Observation (2/2)

- Aneris Distributed Separation Logic
  > provides rules to reason about unreliable unconnected communication;
  > had no native/library support for reliable/connected communication (i.e. each time reliability/sessions had to be built in ad-hoc way).

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

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

(c) socket protocol predicate \( sa \Rightarrow \Phi \)
Let Aneris and Actris projects meet to enable reasoning about reliable network communication!

...The rendez-vous point is our verified client-server library.
II. 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

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

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

How **server** deserialises values received from the **client**

```ocaml
open Ast

type ('a, 'b) client_sktt
type ('a, 'b) server_sktt
type ('a, 'b) chan_descr

val make_client_sktt : ('a of 'b) serialzier -> ('b of 'a serialzier -> saddrr -> ('a, 'b) client_sktt
val make_server_sktt : ('a of 'b) serialzier -> ('b of 'a serialzier -> saddrr -> ('a, 'b) server_sktt
val server_listen : ('a, 'b) server_sktt -> unit
val accept : ('a, 'b) server_sktt -> ('a, 'b) chan_descr * saddrr
val connect : ('a, 'b) client_sktt -> saddrr -> ('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 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** deserialises values received from the **server**
Example: echo server

```ocaml
open Ast
open Serialization_code
open Client_server_code

let int_s = int_serialzier
let str_s = string_serialzier

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_skkt int_s str_s srv in
  server_listen s;
  fork accept_loop s

let client clt srv s1 s2 =
  let s = make_client_skkt 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"
```
III. Specification
Spec (1/4): Parameters & Resources

User Parameters:

\[ \text{UserParams} \triangleq \begin{cases} \text{srv} : \text{Address}; & \text{srv\_ser} : \text{Serialization}; \\ \text{prot} : \text{iProto}; & \text{clt\_ser} : \text{Serialization}; \end{cases} \]

Session Resources:

\[ \text{SessionResources(UP : UserParams)} \triangleq \begin{cases} \text{srv\_si} : \text{Message} \rightarrow \text{iProp}; & \text{CanConnect} : \text{Val} \rightarrow \text{Address}; \rightarrow \text{iProp}; \\ \text{SrvInit} : \text{iProp}; & c \xrightarrow{\text{ip}_{\text{ser}}} !\vec{x} : \vec{r} \langle v \rangle \{P\}. \text{prot} \quad (\text{mapsto connective}); \\ \text{CanListen} : \text{Val} \rightarrow \text{iProp}; & \text{laws about those resources} \quad (\text{e.g. subprotocols}); \\ \text{Listens} : \text{Val} \rightarrow \text{iProp}; \end{cases} \]

Notations: \[ S := \text{SessionResources(UP)}, \ S.\text{srv} := \text{UP.srv} \]
Client Setup:

\[
\text{HT-make-client-socket}
\{ \text{FreeAddr(clt)} \times \text{clt} \leadsto (\emptyset, \emptyset) \times \text{clt} \neq S.\text{srv} \}
\langle \text{clt}_{\text{ip}}; \text{mk}_\text{clt}_\text{skt} (S.\text{srv}_\text{ser}) (S.\text{clt}_\text{ser}) \text{clt} \rangle
\{ w. \exists \text{skt}. w = \text{skt} \times S.\text{CanConnect clt skt} \}
\]

\[
\text{HT-connect}
\{ S.\text{CanConnect clt skt} \}
\langle \text{clt}_{\text{ip}}; \text{connect skt} S.\text{srv} \rangle
\{ w. \exists c. w = c \times c \xrightarrow[clt_{ip}]{S.\text{clt}_\text{ser}} S.\text{prot} \}
\]

Server Setup:

\[
\text{HT-make-server-socket}
\{ \text{FreeAddr}(S.\text{srv}) \times S.\text{srv} \leadsto (\emptyset, \emptyset) \times S.\text{srv} \Rightarrow S.\text{srv}_\text{si} \times S.\text{SrvInit} \}
\langle S.\text{srv}_{\text{ip}}; \text{mk}_\text{srv}_\text{skt} S.\text{srv}_\text{ser} S.\text{clt}_\text{ser} S.\text{srv} \rangle
\{ w. \exists \text{skt}. w = \text{skt} \times S.\text{CanListen skt} \}
\]

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

\[
\text{HT-accept}
\{ S.\text{Listens skt} \} \langle S.\text{srv}_{\text{ip}}; \text{accept skt} \rangle \{ w. \exists c. w = (c, sa) \times S.\text{Listens skt} \times c \xrightarrow[S.\text{srv}_{\text{ip}}]{S.\text{srv}_\text{ser}} S.\text{prot} \}
\]
Spec (2/4): Client/Server Setup

### Client Setup:

\texttt{ht-make-client-socket}

\[
\{ \text{FreeAddr}(clt) \times clt \sim (\emptyset, \emptyset) \times clt \neq S.srv \} \\
\langle clt_{ip}, mk\_clt\_skt(S.srv\_ser)(S.clt\_ser)clt \rangle \\
\{ w. \exists skt. w = skt \times S.\text{CanConnect} clt skt \}\]

\texttt{ht-connect}

\[
\{ S.\text{CanConnect} clt skt \} \\
\langle clt_{ip}; \text{connect skt} S.srv \rangle \\
\{ w. \exists c. w = c \times c \xrightarrow{clt_{ip}} S.prot \}
\]

### Server Setup:

\texttt{ht-make-server-socket}

\[
\{ \text{FreeAddr}(S.srv) \times S.srv \sim (\emptyset, \emptyset) \times \} \\
\{ S.srv \Rightarrow S.srv\_si \times S.SrvInit \} \\
\langle S.srv_{ip} ; mk\_srv\_skt S.srv\_ser S.clt\_ser S.srv \rangle \\
\{ w. \exists skt. w = skt \times S.\text{CanListen} skt \} \]

\texttt{ht-listen}

\[
\{ S.\text{CanListen} skt \} \\
\langle S.srv_{ip} ; \text{listen skt} \rangle \\
\{ S.\text{Listens} skt \}
\]

\texttt{ht-accept}

\[
\{ S.\text{Listens} skt \} \langle S.srv_{ip}; \text{accept skt} \rangle \{ w. \exists c. w = (c, sa) \times S.\text{Listens} skt \times c \xrightarrow{S.srv_{ip}} S.prot \}
\]
**Spec (3/4): Reliable Data Transfer**

\[\text{HT-reliable-recv}\]
\[
\{c \xrightarrow{ip\_ser} ?\tilde{x}:\tilde{t}\langle v \rangle\{P\}. \text{prot} \langle ip; \text{recv} c \rangle \{w. \exists \tilde{y}. w = v[\tilde{y}/\tilde{x}] \ast c \xrightarrow{ip\_ser} \text{prot}[\tilde{y}/\tilde{x}] \ast P[\tilde{y}/\tilde{x}]\}\}
\]

\[\text{HT-reliable-send}\]
\[
\{c \xrightarrow{ip\_ser} !\tilde{x}:\tilde{t}\langle v \rangle\{P\}. \text{prot} \ast P[\tilde{t}/\tilde{x}] \ast \text{Ser} \text{ser } (v[\tilde{t}/\tilde{x}])\langle ip; \text{send} c (v[\tilde{t}/\tilde{x}]\rangle \{c \xrightarrow{ip\_ser} \text{prot}[\tilde{t}/\tilde{x}]\}\}
\]
**Init-setup**

\[ \text{True} \Rightarrow \exists S : \text{SessionResources}(\text{UP}). \]

\[ S.SrvInit * \]

\[ (\forall sa, \text{HT-make-client-socket}[S](sa)) * \]

\[ \text{HT-make-server-socket}[S] * \]

\[ (\forall skt sa, \text{HT-connect}[S](skt, sa)) * \]

\[ (\text{specs for listen, accept, send, recv, try_recv}) \]
let rec echo_loop c =
    let req = recv c in
    send c (strlen req);
    echo_loop c

OCaml function

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

Generated Coq definition

Definition prot_aux (rec : iProto Σ) : iProto Σ :=
    (<! (s : string)> MSG #s ; <? (n : ℕ) > MSG #n {{ r.String.length s = n }}); rec%proto.

Protocol

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 HΦ". iLöb as "IH". wp_lam.
    wp_recv (s₁) as "_". wp_send with "[//]".
    wp_seq.by iApply ("IH" with "[$Hci]").

Qed.

Proof of echo_loop
IV. Verification
The implementation of send and recv is the same for client and server. In fact, their implementation is also agnostic of network.

This is possible because channels are using in- and out- buffers as indirection (calling send enqueues to the out-buffer, calling recv dequeues from the in-buffer).
The rendez-vous point

Crucially, this is also where the connection between Actris Ghost Theory and the implementation takes place. However, this connection is not immediate:

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

- **the four physical buffers**
  - play different role (out-buffer simply (re)transmits, in-buffer keeps data for delivery)
  - are local data of each node and are updated asynchronously
More buffers, seriously?

- Our solution is to 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** must hold between Actris, glue, and physical buffers:
  - $R_r$ is prefix of $T_l$ and $R_l$ is prefix of $T_r$  \(\text{(Internal-Coh)}\)
  - $v_1 = T_l - R_r$ and $v_2 = T_r - R_l$ \(\text{(Actris-Coh)}\)
  - $sbufl$ is suffix of $T_l$ and $sbufr$ is suffix of $T_r$ \(\text{(SBuf-Coh)}\)
  - $rbufl$ is prefix of $(T_r - R_l)$ and $rbufr$ is prefix of $(T_l - R_r)$ \(\text{(Rbuf-Coh)}\)

- The **verification** is then primarily an effort in **preserving these relations**, in the presence of the concurrent accesses of the communication layer.
Other Observations (1/3)

- 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
Client Implementation

Connection Opening

Data Transfer

User Called Methods

Internal State

Internal Procedures

Network Communication