Modelling and Verifying Distributed Systems
with Aneris

Jonas Kastberg Hinrichsen, Aarhus University

and
the Aneris team

24. October 2022
Aarhus University

Verification of Distributed Systems

Distributed systems are a suitable target for formal verification

- More relevant than ever: Cloud Computing, Internet of Things, Mobile devices.
- Hard to get right: Even simple programs can have non-trivial bugs
- Important to get right: Even minor bugs can have large implications

Verification of Distributed Systems

Distributed systems are a suitable target for formal verification

- More relevant than ever: Cloud Computing, Internet of Things, Mobile devices.
- Hard to get right: Even simple programs can have non-trivial bugs
- Important to get right: Even minor bugs can have large implications
- Doubly so in an unreliable network (e.g. UDP-based)
  - **Unreliable:** Messages can be dropped, duplicated, and reordered

Verification of Distributed Systems

Distributed systems are a suitable target for formal verification

- More relevant than ever: Cloud Computing, Internet of Things, Mobile devices.
- Hard to get right: Even simple programs can have non-trivial bugs
- Important to get right: Even minor bugs can have large implications
- Doubly so in an unreliable network (e.g. UDP-based)
  - **Unreliable:** Messages can be dropped, duplicated, and reordered

We can carry out such formal verification as we have:

- Distributed Semantics
- Distributed Program Logic

Verification of Distributed Systems

Distributed systems are a suitable target for formal verification

- More relevant than ever: Cloud Computing, Internet of Things, Mobile devices.
- Hard to get right: Even simple programs can have non-trivial bugs
- Important to get right: Even minor bugs can have large implications
- Doubly so in an unreliable network (e.g. UDP-based)
  - **Unreliable:** Messages can be dropped, duplicated, and reordered

We can carry out such formal verification as we have:

- Distributed Semantics: **AnerisLang**, an OCaml-like language with UDP sockets
- Distributed Program Logic

Verification of Distributed Systems

Distributed systems are a suitable target for formal verification

- More relevant than ever: Cloud Computing, Internet of Things, Mobile devices.
- Hard to get right: Even simple programs can have non-trivial bugs
- Important to get right: Even minor bugs can have large implications
- Doubly so in an unreliable network (e.g. UDP-based)
  - **Unreliable:** Messages can be dropped, duplicated, and reordered

We can carry out such formal verification as we have:

- Distributed Semantics: **AnerisLang**, an OCaml-like language with UDP sockets
- Distributed Program Logic: **Aneris**, a program logic for AnerisLang in Iris

Overview

The **AnerisLang** Distributed Semantics

- Modelling unreliable distributed networks
- Examples of distributed programs
- Pitfalls of unreliable communication

The **Aneris** Distributed Program Logic

- Properties of a distributed program logic
- Modular reasoning principles of unreliable distributed systems
- Examples of verification with the logic

Distributed Semantics

Distributed Semantics

A distributed semantics should consider:

1. The semantics of the individual nodes
   - To capture network-sensitive node-local behaviour

Distributed Semantics

A distributed semantics should consider:

1. The semantics of the individual nodes
   - To capture network-sensitive node-local behaviour
2. The semantics of the network connectives
   - Socket allocation and binding
   - Message sending and receiving

Distributed Semantics

A distributed semantics should consider:

1. The semantics of the individual nodes
   - To capture network-sensitive node-local behaviour
2. The semantics of the network connectives
   - Socket allocation and binding
   - Message sending and receiving
3. The semantics of the (unreliable) network
   - Delivery of messages
   - Dropping, duplication, and reordering of messages

Distributed Semantics

A distributed semantics should consider:

1. The semantics of the individual nodes
   - To capture network-sensitive node-local behaviour
2. The semantics of the network connectives
   - Socket allocation and binding
   - Message sending and receiving
3. The semantics of the (unreliable) network
   - Delivery of messages
   - Dropping, duplication, and reordering of messages
4. A notion of safety
   - e.g. e is safe if it never gets stuck

AnerisLang: an OCaml-like language with UDP sockets

Node-local semantics designed to be as close to OCaml as possible:

$$
\begin{aligned}
v \in \mathit{Val} &::= () \mid b \mid i \mid s \mid \ell \mid \texttt{rec}\ f\ x = e \mid \ldots \\
e \in \mathit{Expr} &::= v \mid x \mid \texttt{rec}\ f\ x = e \mid e_1\ e_2 \mid \texttt{ref}(e) \mid !\,e \mid e_1 \leftarrow e_2 \mid \\
&\qquad \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{assert}\ e \mid \texttt{fork}\,(e) \mid \ldots
\end{aligned}
$$

AnerisLang: an OCaml-like language with UDP sockets

Node-local semantics designed to be as close to OCaml as possible:

$$
\begin{aligned}
v \in \mathit{Val} &::= () \mid b \mid i \mid s \mid \ell \mid \texttt{rec}\, f\, x = e \mid \ldots \\
e \in \mathit{Expr} &::= v \mid x \mid \texttt{rec}\, f\, x = e \mid e_1\ e_2 \mid \texttt{ref}(e) \mid !\, e \mid e_1 \leftarrow e_2 \mid \\
&\qquad \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \mid \texttt{assert}\ e \mid \texttt{fork}\,(e) \mid \ldots
\end{aligned}
$$

Socket semantics inspired by UDP sockets:

$$
\begin{aligned}
v \in \mathit{Val} &::= \ldots \mid \mathit{sh} \mid \mathit{sa} \mid \ldots \\
e \in \mathit{Expr} &::= \ldots \mid \texttt{socket} \mid \texttt{socketbind}\ e_1\ e_2 \mid \texttt{send}\ e_1\ e_2\ e_3 \mid \texttt{recv}\ e \mid \ldots
\end{aligned}
$$

AnerisLang: an OCaml-like language with UDP sockets

Node-local semantics designed to be as close to OCaml as possible:

$$
\begin{aligned}
v \in \mathit{Val} &::= () \mid b \mid i \mid s \mid \ell \mid \texttt{rec}\ f\ x = e \mid \ldots \\
e \in \mathit{Expr} &::= v \mid x \mid \texttt{rec}\ f\ x = e \mid e_1\ e_2 \mid \texttt{ref}(e) \mid !\,e \mid e_1 \leftarrow e_2 \mid \\
&\qquad \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{assert}\ e \mid \texttt{fork}\,(e) \mid \ldots
\end{aligned}
$$

Socket semantics inspired by UDP sockets:

$$
\begin{aligned}
v \in \mathit{Val} &::= \ldots \mid \mathit{sh} \mid \mathit{sa} \mid \ldots \\
e \in \mathit{Expr} &::= \ldots \mid \texttt{socket} \mid \texttt{socketbind}\ e_1\ e_2 \mid \texttt{send}\ e_1\ e_2\ e_3 \mid \texttt{recv}\ e \mid \ldots
\end{aligned}
$$

Network semantics are unreliable:

- Network arbitrarily takes steps alongside nodes
- Network steps may drop, duplicate, or reorder messages in transit

AnerisLang: an OCaml-like language with UDP sockets

Node-local semantics designed to be as close to OCaml as possible:

$$
\begin{aligned}
v \in \mathit{Val} &::= () \mid b \mid i \mid s \mid \ell \mid \texttt{rec}\, f\, x = e \mid \ldots \\
e \in \mathit{Expr} &::= v \mid x \mid \texttt{rec}\, f\, x = e \mid e_1\ e_2 \mid \texttt{ref}(e) \mid !\, e \mid e_1 \leftarrow e_2 \mid \\
&\qquad \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \mid \texttt{assert}\ e \mid \texttt{fork}\,(e) \mid \ldots
\end{aligned}
$$

Socket semantics inspired by UDP sockets:

$$
\begin{aligned}
v \in \mathit{Val} &::= \ldots \mid \mathit{sh} \mid \mathit{sa} \mid \ldots \\
e \in \mathit{Expr} &::= \ldots \mid \texttt{socket} \mid \texttt{socketbind}\ e_1\ e_2 \mid \texttt{send}\ e_1\ e_2\ e_3 \mid \texttt{recv}\ e \mid \ldots
\end{aligned}
$$

Network semantics are unreliable:

- Network arbitrarily takes steps alongside nodes
- Network steps may drop, duplicate, or reorder messages in transit

Safety defined as programs not getting stuck

An example: Ping Pong Service

The server exposes a ping pong service on the address sa_{pong}.
The client uses the service once by sending "ping" and awaiting "pong".

$\texttt{clt}_{\text{pong}}\ sa \triangleq$
  `let` $sh =$ `socket in`
  `socketbind` $sh\ sa$;
  `send` sh "Ping" sa_{pong};
  `let` $m =$ `recv` sh `in`
  `assert` (`fst` $m =$ "Pong")

$\texttt{srv}_{\text{pong}} \triangleq$
  `let` $sh =$ `socket in`
  `socketbind` $sh\ sa_{\text{pong}}$;
  `rec` $go\ \_ =$
    (`let` $m =$ `recv` sh `in`
     `if fst` $m =$ "Ping"
     `then send` sh "Pong" (`snd` m); go ()
     `else assert false`) ()

An example: Ping Pong Service

The server exposes a ping pong service on the address sa_{pong}.
The client uses the service once by sending "ping" and awaiting "pong".

```
clt_pong sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong")
```

```
srv_pong ≜
  let sh = socket in
  socketbind sh sa_pong;
  rec go _ =
    (let m = recv sh in
     if fst m = "Ping"
     then send sh "Pong" (snd m); go ()
     else assert false) ()
```

It this safe?

An example: Ping Pong Service

The server exposes a ping pong service on the address sa_{pong}.
The client uses the service once by sending "ping" and awaiting "pong".

```
clt_pong sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong")
```

```
srv_pong ≜
  let sh = socket in
  socketbind sh sa_pong;
  rec go _ =
    (let m = recv sh in
     if fst m = "Ping"
     then send sh "Pong" (snd m); go ()
     else assert false) ()
```

It this safe?

- What if messages are dropped?

An example: Ping Pong Service

The server exposes a ping pong service on the address sa_{pong}.
The client uses the service once by sending "ping" and awaiting "pong".

```
clt_pong sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong")
```

```
srv_pong ≜
  let sh = socket in
  socketbind sh sa_pong;
  rec go _ =
    (let m = recv sh in
     if fst m = "Ping"
     then send sh "Pong" (snd m); go ()
     else assert false) ()
```

It this safe?

- What if messages are dropped?
- What if messages are duplicated?

An example: Ping Pong Service

The server exposes a ping pong service on the address sa_{pong}.
The client uses the service once by sending "ping" and awaiting "pong".

```
clt_pong sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong")
```

```
srv_pong ≜
  let sh = socket in
  socketbind sh sa_pong;
  rec go _ =
    (let m = recv sh in
     if fst m = "Ping"
     then send sh "Pong" (snd m); go ()
     else assert false) ()
```

It this safe?

- What if messages are dropped?
- What if messages are duplicated?
- What if messages are reordered?

An example: Ping Pong Service

The server exposes a ping pong service on the address sa_{pong}.
The client uses the service once by sending "ping" and awaiting "pong".

```
clt_pong sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong")
```

```
srv_pong ≜
  let sh = socket in
  socketbind sh sa_pong;
  rec go _ =
    (let m = recv sh in
     if fst m = "Ping"
     then send sh "Pong" (snd m); go ()
     else assert false) ()
```

It this safe?

- What if messages are dropped? (safe, but loops ✓)
- What if messages are duplicated?
- What if messages are reordered?

An example: Ping Pong Service

The server exposes a ping pong service on the address sa_{pong}.
The client uses the service once by sending "ping" and awaiting "pong".

```
clt_pong sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong")
```

```
srv_pong ≜
  let sh = socket in
  socketbind sh sa_pong;
  rec go _ =
    (let m = recv sh in
     if fst m = "Ping"
     then send sh "Pong" (snd m); go ()
     else assert false) ()
```

It this safe?

- What if messages are dropped? (safe, but loops ✓)
- What if messages are duplicated? (safe, as server just keeps responding ✓)
- What if messages are reordered?

An example: Ping Pong Service

The server exposes a ping pong service on the address sa_{pong}.
The client uses the service once by sending "ping" and awaiting "pong".

```
clt_pong sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong")
```

```
srv_pong ≜
  let sh = socket in
  socketbind sh sa_pong;
  rec go _ =
    (let m = recv sh in
     if fst m = "Ping"
     then send sh "Pong" (snd m); go ()
     else assert false) ()
```

It this safe?

- What if messages are dropped? (safe, but loops ✓)
- What if messages are duplicated? (safe, as server just keeps responding ✓)
- What if messages are reordered? (safe, as all directed messages are the same ✓)

Another example: Echo Service

The server exposes an echo service on the address sa_{echo}.
The client uses the service twice, first sending "Hello" and then "World".

$\mathtt{clt}_{\mathtt{echo}}\ sa \triangleq$
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_{echo};
  send sh "World" sa_{echo};
  let m_1 = recv sh in
  let m_2 = recv sh in
  assert (fst m_1 = "Hello");
  assert (fst m_2 = "World")

$\mathtt{srv}_{\mathtt{echo}} \triangleq$
  let sh = socket in
  socketbind sh sa_{echo};
  rec go _ =
    (let m = recv sh in
     send sh (fst m) (snd m);
     go ()) ()

Another example: Echo Service

The server exposes an echo service on the address sa_{echo}.
The client uses the service twice, first sending "Hello" and then "World".

```
clt_echo sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  send sh "World" sa_echo;
  let m1 = recv sh in
  let m2 = recv sh in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

```
srv_echo ≜
  let sh = socket in
  socketbind sh sa_echo;
  rec go _ =
    (let m = recv sh in
     send sh (fst m) (snd m);
     go ()) ()
```

Is this safe?

Another example: Echo Service

The server exposes an echo service on the address sa_{echo}.
The client uses the service twice, first sending "Hello" and then "World".

$\texttt{clt}_{\texttt{echo}}\ sa \triangleq$
$\quad \texttt{let}\ sh = \texttt{socket}\ \texttt{in}$
$\quad \texttt{socketbind}\ sh\ sa;$
$\quad \texttt{send}\ sh\ \text{"Hello"}\ sa_{echo};$
$\quad \texttt{send}\ sh\ \text{"World"}\ sa_{echo};$
$\quad \texttt{let}\ m_1 = \texttt{recv}\ sh\ \texttt{in}$
$\quad \texttt{let}\ m_2 = \texttt{recv}\ sh\ \texttt{in}$
$\quad \texttt{assert}\ (\texttt{fst}\ m_1 = \text{"Hello"});$
$\quad \texttt{assert}\ (\texttt{fst}\ m_2 = \text{"World"})$

$\texttt{srv}_{\texttt{echo}} \triangleq$
$\quad \texttt{let}\ sh = \texttt{socket}\ \texttt{in}$
$\quad \texttt{socketbind}\ sh\ sa_{echo};$
$\quad \texttt{rec}\ go\ \_ =$
$\quad\quad (\texttt{let}\ m = \texttt{recv}\ sh\ \texttt{in}$
$\quad\quad\ \texttt{send}\ sh\ (\texttt{fst}\ m)\ (\texttt{snd}\ m);$
$\quad\quad\ go\ ())\ ()$

Is this safe?

- What if message are dropped?

Another example: Echo Service

The server exposes an echo service on the address sa_{echo}.
The client uses the service twice, first sending "Hello" and then "World".

```
clt_echo sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  send sh "World" sa_echo;
  let m1 = recv sh in
  let m2 = recv sh in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

```
srv_echo ≜
  let sh = socket in
  socketbind sh sa_echo;
  rec go _ =
    (let m = recv sh in
     send sh (fst m) (snd m);
     go ()) ()
```

Is this safe?

- What if message are dropped?
- What if messages are duplicated?

Another example: Echo Service

The server exposes an echo service on the address sa_{echo}.
The client uses the service twice, first sending "Hello" and then "World".

$\texttt{clt}_{\texttt{echo}}\ sa \triangleq$
  `let` $sh =$ `socket in`
  `socketbind` $sh\ sa$;
  `send` sh "Hello" sa_{echo};
  `send` sh "World" sa_{echo};
  `let` $m_1 =$ `recv` sh `in`
  `let` $m_2 =$ `recv` sh `in`
  `assert` (`fst` $m_1 =$ "Hello");
  `assert` (`fst` $m_2 =$ "World")

$\texttt{srv}_{\texttt{echo}} \triangleq$
  `let` $sh =$ `socket in`
  `socketbind` $sh\ sa_{echo}$;
  `rec` $go\ \_ =$
    (`let` $m =$ `recv` sh `in`
     `send` sh (`fst` m) (`snd` m);
     go ()) ()

Is this safe?

- What if message are dropped?
- What if messages are duplicated?
- What if messages are reordered?

Another example: Echo Service

The server exposes an echo service on the address sa_{echo}.
The client uses the service twice, first sending "Hello" and then "World".

$\mathtt{clt}_{\mathtt{echo}}\ sa \triangleq$
  `let` $sh =$ `socket in`
  `socketbind` $sh\ sa$;
  `send` sh "Hello" sa_{echo};
  `send` sh "World" sa_{echo};
  `let` $m_1 =$ `recv` sh `in`
  `let` $m_2 =$ `recv` sh `in`
  `assert` (`fst` $m_1 =$ "Hello");
  `assert` (`fst` $m_2 =$ "World")

$\mathtt{srv}_{\mathtt{echo}} \triangleq$
  `let` $sh =$ `socket in`
  `socketbind` $sh\ sa_{echo}$;
  `rec` $go\ \_ =$
    (`let` $m =$ `recv` sh `in`
     `send` sh (`fst` m) (`snd` m);
     go ()) ()

Is this safe?

- What if message are dropped? (safe, but loops ✓)
- What if messages are duplicated?
- What if messages are reordered?

Another example: Echo Service

The server exposes an echo service on the address sa_{echo}.
The client uses the service twice, first sending "Hello" and then "World".

$\texttt{clt}_{\texttt{echo}}\ sa \triangleq$
$\quad \texttt{let}\ sh = \texttt{socket}\ \texttt{in}$
$\quad \texttt{socketbind}\ sh\ sa;$
$\quad \texttt{send}\ sh\ \text{"Hello"}\ sa_{echo};$
$\quad \texttt{send}\ sh\ \text{"World"}\ sa_{echo};$
$\quad \texttt{let}\ m_1 = \texttt{recv}\ sh\ \texttt{in}$
$\quad \texttt{let}\ m_2 = \texttt{recv}\ sh\ \texttt{in}$
$\quad \texttt{assert}\ (\texttt{fst}\ m_1 = \text{"Hello"});$
$\quad \texttt{assert}\ (\texttt{fst}\ m_2 = \text{"World"})$

$\texttt{srv}_{\texttt{echo}} \triangleq$
$\quad \texttt{let}\ sh = \texttt{socket}\ \texttt{in}$
$\quad \texttt{socketbind}\ sh\ sa_{echo};$
$\quad \texttt{rec}\ go\ \_ =$
$\quad\quad (\texttt{let}\ m = \texttt{recv}\ sh\ \texttt{in}$
$\quad\quad\ \texttt{send}\ sh\ (\texttt{fst}\ m)\ (\texttt{snd}\ m);$
$\quad\quad\ go\ ())\ ()$

Is this safe?

- What if message are dropped? (safe, but loops ✓)
- What if messages are duplicated? (unsafe! may receive "Hello" twice ✗)
- What if messages are reordered?

Another example: Echo Service

The server exposes an echo service on the address sa_{echo}.
The client uses the service twice, first sending "Hello" and then "World".

```
clt_echo sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  send sh "World" sa_echo;
  let m1 = recv sh in
  let m2 = recv sh in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

```
srv_echo ≜
  let sh = socket in
  socketbind sh sa_echo;
  rec go _ =
    (let m = recv sh in
     send sh (fst m) (snd m);
     go ()) ()
```

Is this safe?

- What if message are dropped? (safe, but loops ✓)
- What if messages are duplicated? (unsafe! may receive "Hello" twice ✗)
- What if messages are reordered? (unsafe! may receive "World" before "Hello" ✗)

Another example: Echo Service

The server exposes an echo service on the address sa_{echo}.
The client uses the service twi

```
recvfresh sh ms ≜
  rec go _ =
    (let m = recv sh in
     if mem m ms then go () else m) ()
```

```
clt_echo sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  send sh "World" sa_echo;
  let m1 = recv sh in
  let m2 = recvfresh sh [m1] in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

```
rec go _ =
  (let m = recv sh in
   send sh (fst m) (snd m);
   go ()) ()
```

Is this safe?

- What if message are dropped? (safe, but loops ✓)
- What if messages are duplicated? (unsafe! may receive "Hello" twice ✗)
- What if messages are reordered? (unsafe! may receive "World" before "Hello" ✗)

Another example: Echo Service

The server exposes an echo service on the address sa_{echo}.
The client uses the service twice, first sending "Hello" and then "World".

```
clt_echo sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  send sh "World" sa_echo;
  let m1 = recv sh in
  let m2 = recvfresh sh [m1] in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

```
srv_echo ≜
  let sh = socket in
  socketbind sh sa_echo;
  rec go _ =
    (let m = recv sh in
     send sh (fst m) (snd m);
     go ()) ()
```

Is this safe?

- What if message are dropped? (safe, but loops ✓)
- What if messages are duplicated? (safe, as we wait for a fresh second message ✓)
- What if messages are reordered? (unsafe! may receive "World" before "Hello" ✗)

Another example: Echo Service

The server exposes an echo service on the address sa_{echo}.
The client uses the service twice, first sending "Hello" and then "World".

```
clt_echo sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  let m1 = recv sh in
  send sh "World" sa_echo;
  let m2 = recvfresh sh [m1] in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

```
srv_echo ≜
  let sh = socket in
  socketbind sh sa_echo;
  rec go _ =
    (let m = recv sh in
     send sh (fst m) (snd m);
     go ()) ()
```

Is this safe?

- What if message are dropped? (safe, but loops ✓)
- What if messages are duplicated? (safe, as we wait for a fresh second message ✓)
- What if messages are reordered? (unsafe! may receive "World" before "Hello" ✗)

Another example: Echo Service

The server exposes an echo service on the address sa_{echo}.
The client uses the service twice, first sending "Hello" and then "World".

```
clt_echo sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  let m1 = recv sh in
  send sh "World" sa_echo;
  let m2 = recvfresh sh [m1] in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

```
srv_echo ≜
  let sh = socket in
  socketbind sh sa_echo;
  rec go _ =
    (let m = recv sh in
     send sh (fst m) (snd m);
     go ()) ()
```

Is this safe?

- What if message are dropped? (safe, but loops ✓)
- What if messages are duplicated? (safe, as we wait for a fresh second message ✓)
- What if messages are reordered? (safe, as we can only receive "Hello" first ✓)

Verification of Distributed Systems

Verification of Distributed Systems

We must be able to reason about:

1. Non-distributed internal node reductions
2. Allocation and binding of sockets
3. Message passing (and resource transfer)

Verification of Distributed Systems

We must be able to reason about:

1. Non-distributed internal node reductions
2. Allocation and binding of sockets
3. Message passing (and resource transfer)

We want to guarantee **safety**

- No node in the distributed system will ever get stuck

Verification of Distributed Systems

We must be able to reason about:

1. Non-distributed internal node reductions
2. Allocation and binding of sockets
3. Message passing (and resource transfer)

We want to guarantee **safety**

- No node in the distributed system will ever get stuck

We want to obtain **abstraction** and **modularity**:

- **Abstraction:** abstract over unreliable network layer
- **Modularity:** reason about nodes individually

Aneris: Distributed Separation Logic

Distributed Program Logic for *AnerisLang*, built on top of Iris, with:

- Node-local Hoare triple rules for non-distributed expressions
- Node-local Hoare triple rules for sockets
- Node-local Hoare triple rules for message passing

Aneris: Distributed Separation Logic

Distributed Program Logic for *AnerisLang*, built on top of Iris, with:

- Node-local Hoare triple rules for non-distributed expressions
- Node-local Hoare triple rules for sockets
- Node-local Hoare triple rules for message passing

Aneris inherits Iris's safety guarantees (which are foundationally certified in Coq)

Node-local rules for non-distributed expressions

Standard rules decorated with an ip identifier:

$$
\begin{aligned}
\tau, \sigma ::= {} & x \mid 0 \mid 1 \mid \mathsf{B} \mid \mathbb{N} \mid \mathsf{Z} \mid \mathsf{Type} \mid \forall x : \tau.\, \sigma \mid \mathit{Loc} \mid \mathit{Val} \mid \mathit{Expr} \mid \mathsf{Prop} \mid \mathit{Ip} \mid \ldots \\
t, u, P, Q ::= {} & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid && \text{(Propositional logic)} \\
& \forall x : \tau.\, P \mid \exists x : \tau.\, P \mid t = u \mid && \text{(Higher-order logic with equality)} \\
& P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \overset{ip}{\mapsto} v \mid \{P\}\, \langle ip;\, e \rangle\, \{v.\, Q\} \mid && \text{(Separation logic)} \\
& \triangleright P \mid \boxed{P} \mid P \Rrightarrow Q \mid \ldots && \text{(Ghost state and invariants)}
\end{aligned}
$$

Node-local rules for non-distributed expressions

Standard rules decorated with an ip identifier:

$$\begin{aligned}
\tau, \sigma ::= {} & x \mid 0 \mid 1 \mid \mathsf{B} \mid \mathbb{N} \mid \mathsf{Z} \mid \mathsf{Type} \mid \forall x : \tau.\, \sigma \mid \mathit{Loc} \mid \mathit{Val} \mid \mathit{Expr} \mid \mathsf{Prop} \mid \mathit{Ip} \mid \ldots \\
t, u, P, Q ::= {} & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid && \text{(Propositional logic)} \\
& \forall x : \tau.\, P \mid \exists x : \tau.\, P \mid t = u \mid && \text{(Higher-order logic with equality)} \\
& P * Q \mid P \mathrel{-\!\!*} Q \mid \ell \overset{ip}{\mapsto} v \mid \{P\}\, \langle ip;\, e \rangle\, \{v.\, Q\} \mid && \text{(Separation logic)} \\
& \triangleright P \mid \boxed{P} \mid P \Rrightarrow Q \mid \ldots && \text{(Ghost state and invariants)}
\end{aligned}$$

Example: rules for references:

HT-ALLOC

$$\{\mathsf{True}\}\, \langle ip;\, \texttt{ref}(v) \rangle\, \{w.\, \exists \ell.\, w = \ell * \ell \overset{ip}{\mapsto} v\}$$

HT-LOAD

$$\{\ell \overset{ip}{\mapsto} v\}\, \langle ip;\, !\,\ell \rangle\, \{w.\, w = v * \ell \overset{ip}{\mapsto} v\}$$

HT-STORE

$$\{\ell \overset{ip}{\mapsto} v\}\, \langle ip;\, \ell \leftarrow w \rangle\, \{\ell \overset{ip}{\mapsto} w\}$$

Node-local rules for sockets

$$\tau, \sigma ::= \ldots \mid \mathit{Socket} \mid \mathit{Address} \mid \ldots$$
$$t, u, P, Q ::= \ldots \mid \mathit{sh} \overset{\mathit{ip}}{\hookrightarrow} o \mid \mathsf{FreeAddr}(\mathit{sa}) \mid \ldots$$

HT-NEWSOCKET

$\{\mathsf{True}\}$
$\quad \langle \mathit{ip};\ \texttt{socket}() \rangle$
$\{w.\ \exists \mathit{sh}.\ w = \mathit{sh} * \mathit{sh} \overset{\mathit{ip}}{\hookrightarrow} \mathsf{None}\}$

HT-SOCKETBIND

$\{\mathit{sh} \overset{\mathit{sa}.\mathtt{ip}}{\hookrightarrow} \mathsf{None} * \mathsf{FreeAddr}(\mathit{sa})\}$
$\quad \langle \mathit{sa}.\mathtt{ip};\ \texttt{socketbind}\ \mathit{sh}\ \mathit{sa} \rangle$
$\{w.\ w = () * \mathit{sh} \overset{\mathit{sa}.\mathtt{ip}}{\hookrightarrow} \mathsf{Some}(\mathit{sa})\}$

Node-local rules for sockets

$$\tau, \sigma ::= \ldots \mid \mathit{Socket} \mid \mathit{Address} \mid \ldots$$
$$t, u, P, Q ::= \ldots \mid \mathit{sh} \overset{\mathit{ip}}{\hookrightarrow} o \mid \mathsf{FreeAddr}(\mathit{sa}) \mid \ldots$$

HT-NEWSOCKET
$\{\mathsf{True}\}$
$\quad \langle \mathit{ip};\ \texttt{socket}() \rangle$
$\{w.\ \exists \mathit{sh}.\ w = \mathit{sh} * \mathit{sh} \overset{\mathit{ip}}{\hookrightarrow} \mathsf{None}\}$

HT-SOCKETBIND
$\{\mathit{sh} \overset{\mathit{sa}.\mathtt{ip}}{\hookrightarrow} \mathsf{None} * \mathsf{FreeAddr}(\mathit{sa})\}$
$\quad \langle \mathit{sa}.\mathtt{ip};\ \texttt{socketbind}\ \mathit{sh}\ \mathit{sa} \rangle$
$\{w.\ w = () * \mathit{sh} \overset{\mathit{sa}.\mathtt{ip}}{\hookrightarrow} \mathsf{Some}(\mathit{sa})\}$

Sockets are treated simlarly to references

- We assume an infinite range, so we can always allocate a fresh one
- Deallocation assumed to be handled by the runtime

Node-local rules for sockets

$$\tau, \sigma ::= \ldots \mid \mathit{Socket} \mid \mathit{Address} \mid \ldots$$
$$t, u, P, Q ::= \ldots \mid \mathit{sh} \overset{\mathit{ip}}{\hookrightarrow} o \mid \mathsf{FreeAddr}(\mathit{sa}) \mid \ldots$$

HT-NEWSOCKET
$\{\mathsf{True}\}$
$\quad \langle \mathit{ip};\ \texttt{socket}() \rangle$
$\{w.\ \exists \mathit{sh}.\ w = \mathit{sh} * \mathit{sh} \overset{\mathit{ip}}{\hookrightarrow} \mathsf{None}\}$

HT-SOCKETBIND
$\{\mathit{sh} \overset{\mathit{sa}.\mathtt{ip}}{\hookrightarrow} \mathsf{None} * \mathsf{FreeAddr}(\mathit{sa})\}$
$\quad \langle \mathit{sa}.\mathtt{ip};\ \texttt{socketbind}\ \mathit{sh}\ \mathit{sa} \rangle$
$\{w.\ w = () * \mathit{sh} \overset{\mathit{sa}.\mathtt{ip}}{\hookrightarrow} \mathsf{Some}(\mathit{sa})\}$

Sockets are treated simlarly to references

- We assume an infinite range, so we can always allocate a fresh one
- Deallocation assumed to be handled by the runtime

All addresses are considered free on node startup

- i.e. the $\mathsf{FreeAddr}(\mathit{sa})$ resource is obtained for any sa for free
- $\mathsf{FreeAddr}(\mathit{sa})$ guarantees that addresses are only bound once

Node-local rules for message passing

Many ways of reasoning about unreliable communication; we want to:

- Transfer (Iris) resources along with messages (to facilitate modularity)
- Abstract over the unreliable semantics (dropping, duplication, and reordering)

Node-local rules for message passing

Many ways of reasoning about unreliable communication; we want to:

- Transfer (Iris) resources along with messages (to facilitate modularity)
- Abstract over the unreliable semantics (dropping, duplication, and reordering)

The **Aneris** solution:

- Treat messages logically as a triple of the source, string, and destination
  - $\mathit{Message} \triangleq \mathit{Address} * \mathit{String} * \mathit{Address}$
  - We often write m.src, m.str, and m.dst for the message components

Node-local rules for message passing

Many ways of reasoning about unreliable communication; we want to:

- Transfer (Iris) resources along with messages (to facilitate modularity)
- Abstract over the unreliable semantics (dropping, duplication, and reordering)

The **Aneris** solution:

- Treat messages logically as a triple of the source, string, and destination
  - $\mathit{Message} \triangleq \mathit{Address} * \mathit{String} * \mathit{Address}$
  - We often write m.src, m.str, and m.dst for the message components
- Associate each socket address with a protocol $\Phi : \mathit{Message} \to \mathit{iProp}$ that all received messages must satisfy
  - Abstracts over reordering, as the order no longer matters

Node-local rules for message passing

Many ways of reasoning about unreliable communication; we want to:

- Transfer (Iris) resources along with messages (to facilitate modularity)
- Abstract over the unreliable semantics (dropping, duplication, and reordering)

The **Aneris** solution:

- Treat messages logically as a triple of the source, string, and destination
  - $\mathit{Message} \triangleq \mathit{Address} * \mathit{String} * \mathit{Address}$
  - We often write m.src, m.str, and m.dst for the message components
- Associate each socket address with a protocol $\Phi : \mathit{Message} \to \mathit{iProp}$ that all received messages must satisfy
  - Abstracts over reordering, as the order no longer matters
- Acquire resources specificed by $\Phi\ m$ only when receiving a *fresh* m
  - Abstracts over duplication, as duplicate messages dont result in duplicate resources

Node-local rules for message passing

Many ways of reasoning about unreliable communication; we want to:

- Transfer (Iris) resources along with messages (to facilitate modularity)
- Abstract over the unreliable semantics (dropping, duplication, and reordering)

The **Aneris** solution:

- Treat messages logically as a triple of the source, string, and destination
  - $\mathit{Message} \triangleq \mathit{Address} * \mathit{String} * \mathit{Address}$
  - We often write m.src, m.str, and m.dst for the message components
- Associate each socket address with a protocol $\Phi : \mathit{Message} \to \mathit{iProp}$ that all received messages must satisfy
  - Abstracts over reordering, as the order no longer matters
- Acquire resources specificed by $\Phi\ m$ only when receiving a *fresh* m
  - Abstracts over duplication, as duplicate messages dont result in duplicate resources
- Require giving up resources specified by $\Phi\ m$ only when sending a *fresh* m
  - Abstracts over dropping, as dropped messages can be retransmitted for free

Node-local rules for message passing

$$\tau, \sigma ::= \ldots \mid \mathit{Message} \mid \ldots$$
$$t, u, P, Q ::= \ldots \mid \mathit{sa} \rightsquigarrow (R, T) \mid \mathit{sa} \Mapsto \Phi \mid \ldots$$

HT-SEND

$$\left\{\begin{array}{l} \mathit{sh} \xhookrightarrow{\mathit{sa}.\mathtt{ip}} \mathrm{Some}(\mathit{sa}) * \mathit{sa} \rightsquigarrow (R, T) * \mathit{dst} \Mapsto \Phi * \\ ((\mathit{sa}, \mathit{str}, \mathit{dst}) \notin T \Rightarrow \Phi\ (\mathit{sa}, \mathit{str}, \mathit{dst})) \end{array}\right\}$$
$$\langle \mathit{sa}.\mathtt{ip};\ \mathtt{send}\ \mathit{sh}\ \mathit{str}\ \mathit{dst} \rangle$$
$$\left\{\begin{array}{l} w.\ w = () * \mathit{sh} \xhookrightarrow{\mathit{sa}.\mathtt{ip}} \mathrm{Some}(\mathit{sa}) * \\ \quad \mathit{sa} \rightsquigarrow (R, T \cup \{(\mathit{sa}, \mathit{str}, \mathit{dst})\}) \end{array}\right\}$$

HT-RECV

$$\left\{\mathit{sh} \xhookrightarrow{\mathit{sa}.\mathtt{ip}} \mathrm{Some}(\mathit{sa}) * \mathit{sa} \rightsquigarrow (R, T) * \mathit{sa} \Mapsto \Phi\right\}$$
$$\langle \mathit{sa}.\mathtt{ip};\ \mathtt{recv}\ \mathit{sh} \rangle$$
$$\left\{\begin{array}{l} w.\ \exists \mathit{str}, \mathit{src}.\ w = (\mathit{str}, \mathit{src}) * \mathit{sh} \xhookrightarrow{\mathit{sa}.\mathtt{ip}} \mathrm{Some}(\mathit{sa}) * \\ \quad \mathit{sa} \rightsquigarrow (R \cup \{(\mathit{src}, \mathit{str}, \mathit{sa})\}, T) * \\ \quad ((\mathit{src}, \mathit{str}, \mathit{sa}) \notin R \Rightarrow \Phi\ (\mathit{src}, \mathit{str}, \mathit{sa})) \end{array}\right\}$$

Node-local rules for message passing

$$\tau, \sigma ::= \ldots \mid \mathit{Message} \mid \ldots$$
$$t, u, P, Q ::= \ldots \mid sa \rightsquigarrow (R, T) \mid sa \Mapsto \Phi \mid \ldots$$

HT-SEND

$$\left\{\begin{array}{l} sh \xhookrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa) * sa \rightsquigarrow (R, T) * dst \Mapsto \Phi * \\ ((sa, str, dst) \notin T \Rightarrow \Phi\ (sa, str, dst)) \end{array}\right\}$$
$$\langle sa.\mathtt{ip};\ \mathtt{send}\ sh\ str\ dst \rangle$$
$$\left\{\begin{array}{l} w.\ w = () * sh \xhookrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa) * \\ \quad sa \rightsquigarrow (R, T \cup \{(sa, str, dst)\}) \end{array}\right\}$$

HT-RECV

$$\left\{sh \xhookrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa) * sa \rightsquigarrow (R, T) * sa \Mapsto \Phi\right\}$$
$$\langle sa.\mathtt{ip};\ \mathtt{recv}\ sh \rangle$$
$$\left\{\begin{array}{l} w.\ \exists str, src.\ w = (str, src) * sh \xhookrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa) * \\ \quad sa \rightsquigarrow (R \cup \{(src, str, sa)\}, T) * \\ \quad ((src, str, sa) \notin R \Rightarrow \Phi\ (src, str, sa)) \end{array}\right\}$$

All addresses have empty histories on node startup

- i.e. the $sa \rightsquigarrow (\emptyset, \emptyset)$ resource is obtained for any sa for free

Node-local rules for message passing

$$\tau, \sigma ::= \ldots \mid \mathit{Message} \mid \ldots$$
$$t, u, P, Q ::= \ldots \mid sa \rightsquigarrow (R, T) \mid sa \Mapsto \Phi \mid \ldots$$

HT-SEND

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa) * sa \rightsquigarrow (R, T) * dst \Mapsto \Phi * \\ ((sa, str, dst) \notin T \Rightarrow \Phi\ (sa, str, dst)) \end{array}\right\}$$
$$\langle sa.\mathtt{ip};\ \mathtt{send}\ sh\ str\ dst \rangle$$
$$\left\{\begin{array}{l} w.\ w = () * sh \xrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa) * \\ \quad sa \rightsquigarrow (R, T \cup \{(sa, str, dst)\}) \end{array}\right\}$$

HT-RECV

$$\left\{sh \xrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa) * sa \rightsquigarrow (R, T) * sa \Mapsto \Phi\right\}$$
$$\langle sa.\mathtt{ip};\ \mathtt{recv}\ sh \rangle$$
$$\left\{\begin{array}{l} w.\ \exists str, src.\ w = (str, src) * sh \xrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa) * \\ \quad sa \rightsquigarrow (R \cup \{(src, str, sa)\}, T) * \\ \quad ((src, str, sa) \notin R \Rightarrow \Phi\ (src, str, sa)) \end{array}\right\}$$

All addresses have empty histories on node startup

- i.e. the $sa \rightsquigarrow (\emptyset, \emptyset)$ resource is obtained for any sa for free

Protocols ($sa \Mapsto \Phi$) are considered either static (for servers) or dynamic (for clients)

- Static protocols are obtained by all nodes on startup

Node-local rules for message passing

$$\tau, \sigma ::= \ldots \mid \mathit{Message} \mid \ldots$$
$$t, u, P, Q ::= \ldots \mid \mathit{sa} \rightsquigarrow (R, T) \mid \mathit{sa} \Mapsto \Phi \mid \texttt{dyn}\ \mathit{sa} \mid \ldots$$

HT-DYNAMIC
$$\frac{\{P * \mathit{sa} \Mapsto \Phi\}\ \langle \mathit{ip};\ e \rangle\ \{Q\}}{\{P * \texttt{dyn}\ \mathit{sa}\}\ \langle \mathit{ip};\ e \rangle\ \{Q\}}$$

HT-SEND
$$\left\{\begin{array}{l} \mathit{sh} \xrightarrow{\mathit{sa}.\mathtt{ip}} \mathrm{Some}(\mathit{sa}) * \mathit{sa} \rightsquigarrow (R, T) * \mathit{dst} \Mapsto \Phi * \\ ((\mathit{sa}, \mathit{str}, \mathit{dst}) \notin T \Rightarrow \Phi\ (\mathit{sa}, \mathit{str}, \mathit{dst})) \end{array}\right\}$$
$$\langle \mathit{sa}.\mathtt{ip};\ \mathtt{send}\ \mathit{sh}\ \mathit{str}\ \mathit{dst} \rangle$$
$$\left\{\begin{array}{l} w.\ w = () * \mathit{sh} \xrightarrow{\mathit{sa}.\mathtt{ip}} \mathrm{Some}(\mathit{sa}) * \\ \quad \mathit{sa} \rightsquigarrow (R, T \cup \{(\mathit{sa}, \mathit{str}, \mathit{dst})\}) \end{array}\right\}$$

HT-RECV
$$\left\{\mathit{sh} \xrightarrow{\mathit{sa}.\mathtt{ip}} \mathrm{Some}(\mathit{sa}) * \mathit{sa} \rightsquigarrow (R, T) * \mathit{sa} \Mapsto \Phi\right\}$$
$$\langle \mathit{sa}.\mathtt{ip};\ \mathtt{recv}\ \mathit{sh} \rangle$$
$$\left\{\begin{array}{l} w.\ \exists \mathit{str}, \mathit{src}.\ w = (\mathit{str}, \mathit{src}) * \mathit{sh} \xrightarrow{\mathit{sa}.\mathtt{ip}} \mathrm{Some}(\mathit{sa}) * \\ \quad \mathit{sa} \rightsquigarrow (R \cup \{(\mathit{src}, \mathit{str}, \mathit{sa})\}, T) * \\ \quad ((\mathit{src}, \mathit{str}, \mathit{sa}) \notin R \Rightarrow \Phi\ (\mathit{src}, \mathit{str}, \mathit{sa})) \end{array}\right\}$$

All addresses have empty histories on node startup

- i.e. the $\mathit{sa} \rightsquigarrow (\emptyset, \emptyset)$ resource is obtained for any sa for free

Protocols ($\mathit{sa} \Mapsto \Phi$) are considered either static (for servers) or dynamic (for clients)

- Static protocols are obtained by all nodes on startup
- Dynamic protocols are obtained via `dyn` sa, given to respective nodes on startup

Verification of the ping pong example - Socket Protocols

```
clt_pong sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong");
```

```
srv_pong ≜
  let sh = socket in
  socketbind sh sa_pong;
  rec go _ =
    (let m = recv sh in
     if fst m = "Ping"
     then send sh "Pong" (snd m); go ()
     else assert false) ()
```

Verification of the ping pong example - Socket Protocols

```
clt_pong sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong");
```

```
srv_pong ≜
  let sh = socket in
  socketbind sh sa_pong;
  rec go _ =
    (let m = recv sh in
     if fst m = "Ping"
     then send sh "Pong" (snd m); go ()
     else assert false) ()
```

Socket protocols:
$\Phi_{clt} \triangleq \lambda m. \ldots$
$\Phi_{srv} \triangleq \lambda m. \ldots$

Verification of the ping pong example - Socket Protocols

```
clt_pong sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong");
```

```
srv_pong ≜
  let sh = socket in
  socketbind sh sa_pong;
  rec go _ =
    (let m = recv sh in
     if fst m = "Ping"
     then send sh "Pong" (snd m); go ()
     else assert false) ()
```

Socket protocols:

$\Phi_{clt} \triangleq \lambda m. \ldots$

$\Phi_{srv} \triangleq \lambda m.\, m.\mathsf{str} = \text{“Ping”} * \ldots$

Verification of the ping pong example - Socket Protocols

```
clt_pong sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong");
```

```
srv_pong ≜
  let sh = socket in
  socketbind sh sa_pong;
  rec go _ =
    (let m = recv sh in
     if fst m = "Ping"
     then send sh "Pong" (snd m); go ()
     else assert false) ()
```

Socket protocols:

$\Phi_{clt} \triangleq \lambda m. \ldots$

$\Phi_{srv} \triangleq \lambda m.\, m.\mathsf{str} = \text{“Ping”} * \exists \psi.\, m.\mathsf{src} \mapsto\!\!\!\Rightarrow \psi * \ldots$

Verification of the ping pong example - Socket Protocols

```
clt_pong sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong");
```

```
srv_pong ≜
  let sh = socket in
  socketbind sh sa_pong;
  rec go _ =
    (let m = recv sh in
     if fst m = "Ping"
     then send sh "Pong" (snd m); go ()
     else assert false) ()
```

Socket protocols:

$\Phi_{clt} \triangleq \lambda m.\ \ldots$

$\Phi_{srv} \triangleq \lambda m.\ m.\mathsf{str} = \text{“Ping”} * \exists \psi.\ m.\mathsf{src} \mapsto\!\Rightarrow \psi * (\forall m'.\ m'.\mathsf{str} = \text{“Pong”} \mathrel{-\!\!*} \psi\ m')$

Verification of the ping pong example - Socket Protocols

```
clt_pong sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong");
```

```
srv_pong ≜
  let sh = socket in
  socketbind sh sa_pong;
  rec go _ =
    (let m = recv sh in
     if fst m = "Ping"
     then send sh "Pong" (snd m); go ()
     else assert false) ()
```

Socket protocols:

$\Phi_{clt} \triangleq \lambda m.\ m.\mathsf{str} = \text{“Pong”}$

$\Phi_{srv} \triangleq \lambda m.\ m.\mathsf{str} = \text{“Ping”} * \exists \psi.\ m.\mathsf{src} \mapsto\!\Rightarrow \psi * (\forall m'.\ m'.\mathsf{str} = \text{“Pong”} \mathrel{-\!\!*} \psi\ m')$

Verification of the ping pong client - Proof

$$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$$
$$\langle sa.\mathtt{ip}; \mathtt{clt}_{\mathrm{pong}}\ sa\rangle$$
$$\{\mathsf{True}\}$$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$

```
  let sh = socket in
  socketbind sh sa;
  send sh "Ping" sa_pong;
  let m = recv sh in
  assert (fst m = "Pong")
```

$\{\mathsf{True}\}$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$
$\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$
$\quad \mathtt{socketbind}\ sh\ sa;$
$\quad \mathtt{send}\ sh\ \text{“Ping”}\ sa_{\mathsf{pong}};$
$\quad \mathtt{let}\ m = \mathtt{recv}\ sh\ \mathtt{in}$
$\quad \mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})$
$\{\mathsf{True}\}$

HT-NEWSOCKET
$\{\mathsf{True}\}$
$\quad \langle ip;\ \mathtt{socket}() \rangle$
$\{w.\ \exists sh.\ w = sh * sh \overset{ip}{\hookrightarrow} \mathsf{None}\}$

HT-SOCKETBIND
$\{sh \overset{sa.\mathtt{ip}}{\hookrightarrow} \mathsf{None} * \mathsf{FreeAddr}(sa)\}$
$\quad \langle sa.\mathtt{ip};\ \mathtt{socketbind}\ sh\ sa \rangle$
$\{w.\ w = () * sh \overset{sa.\mathtt{ip}}{\hookrightarrow} \mathsf{Some}(sa)\}$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$
$\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$
$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$
$\quad \mathtt{socketbind}\ sh\ sa;$
$\quad \mathtt{send}\ sh\ \text{“Ping”}\ sa_{\mathrm{pong}};$
$\quad \mathtt{let}\ m = \mathtt{recv}\ sh\ \mathtt{in}$
$\quad \mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})$
$\{\mathsf{True}\}$

HT-NEWSOCKET
$\{\mathsf{True}\}$
$\quad \langle ip;\ \mathtt{socket}() \rangle$
$\{w.\ \exists sh.\ w = sh * sh \xhookrightarrow{ip} \mathsf{None}\}$

HT-SOCKETBIND
$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa)\}$
$\quad \langle sa.\mathtt{ip};\ \mathtt{socketbind}\ sh\ sa \rangle$
$\{w.\ w = () * sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa)\}$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$
$\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$
$\{sh \xrightarrow{sa.\mathsf{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$
$\quad \mathtt{socketbind}\ sh\ sa;$
$\{sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$
$\quad \mathtt{send}\ sh\ \text{“Ping”}\ sa_{\mathsf{pong}};$
$\quad \mathtt{let}\ m = \mathtt{recv}\ sh\ \mathtt{in}$
$\quad \mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})$
$\{\mathsf{True}\}$

HT-NEWSOCKET
$\{\mathsf{True}\}$
$\quad \langle ip;\ \mathtt{socket}() \rangle$
$\{w.\ \exists sh.\ w = sh * sh \xhookrightarrow{ip} \mathsf{None}\}$

HT-SOCKETBIND
$\{sh \xrightarrow{sa.\mathsf{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa)\}$
$\quad \langle sa.\mathtt{ip};\ \mathtt{socketbind}\ sh\ sa \rangle$
$\{w.\ w = () * sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa)\}$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$
  `let` $sh =$ `socket in`
$\{sh \xrightarrow{sa.\mathrm{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$
  `socketbind` $sh\ sa$;
$\{sh \xrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$
  `send` sh "Ping" sa_{pong};
  `let` $m =$ `recv` sh `in`
  `assert` (`fst` $m =$ "Pong")
$\{\mathsf{True}\}$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$
  `let` $sh =$ `socket in`
$\{sh \xhookrightarrow{sa.\mathtt{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$
  `socketbind` $sh\ sa$;
$\{sh \xhookrightarrow{sa.\mathtt{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$
  `send` sh "Ping" sa_{pong};
  `let` $m =$ `recv` sh `in`
  `assert` (`fst` $m =$ "Pong")
$\{\mathsf{True}\}$

HT-SEND
$\{sh \xhookrightarrow{sa.\mathtt{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (R, T) * dst \Rrightarrow \Phi * ((sa, str, dst) \notin T \Rightarrow \Phi\ (sa, str, dst))\}$
  $\langle sa.\mathtt{ip};$ `send` $sh\ str\ dst\rangle$
$\{w.\ w = () * sh \xhookrightarrow{sa.\mathtt{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (R, T) \cup \{(sa, str, dst)\}\}$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$
  `let sh = socket in`
$\{sh \xrightarrow{sa.\mathsf{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$
  `socketbind sh sa;`
$\{sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$
  `send sh` “Ping” sa_{pong};
  `let m = recv sh in`
  `assert (fst m =` “Pong”`)`
$\{\mathsf{True}\}$

$\Phi_{srv}\ (sa, \text{“Ping”}, sa_{\mathsf{pong}})$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Mapsto \Phi_{srv}\}$
  `let` $sh =$ `socket in`
$\{sh \xrightarrow{sa.\mathsf{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Mapsto \Phi_{srv}\}$
  `socketbind` $sh\ sa$;
$\{sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Mapsto \Phi_{srv}\}$
  `send` sh "Ping" sa_{pong};
  `let` $m =$ `recv` sh `in`
  `assert` (`fst` $m =$ "Pong")
$\{\mathsf{True}\}$

"Ping" $=$ "Ping" $* \exists \psi.\ sa \Mapsto \psi * (\forall m'.\ m'.\mathsf{str} =$ "Pong" $\mathrel{-\!\!*}\ \psi\ m')$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Mapsto \Phi_{srv}\}$
  `let sh = socket in`
$\{sh \xrightarrow{sa.\mathsf{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Mapsto \Phi_{srv}\}$
  `socketbind sh sa;`
$\{sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Mapsto \Phi_{srv}\}$
  `send sh` “Ping” sa_{pong};
  `let m = recv sh in`
  `assert (fst m =` “Pong”`)`
$\{\mathsf{True}\}$

HT-DYNAMIC

$$\frac{\{P * sa \Mapsto \Phi\}\ \langle ip;\ e\rangle\ \{Q\}}{\{P * \mathtt{dyn}\ sa\}\ \langle ip;\ e\rangle\ \{Q\}}$$

“Ping” $=$ “Ping” $* \exists\psi.\ sa \Mapsto \psi * (\forall m'.\ m'.\mathsf{str} =$ “Pong” $\mathrel{-\!\!*}\ \psi\ m')$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Mapsto \Phi_{srv}\}$
  `let` $sh =$ `socket in`
$\{sh \xrightarrow{sa.\mathsf{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Mapsto \Phi_{srv}\}$
  `socketbind` $sh\ sa$;
$\{sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Mapsto \Phi_{clt} * sa_{\mathsf{pong}} \Mapsto \Phi_{srv}\}$
  `send` sh "Ping" sa_{pong};
  `let` $m =$ `recv` sh `in`
  `assert` (`fst` $m =$ "Pong")
$\{\mathsf{True}\}$

HT-DYNAMIC

$$\frac{\{P * sa \Mapsto \Phi\}\ \langle ip;\ e\rangle\ \{Q\}}{\{P * \mathtt{dyn}\ sa\}\ \langle ip;\ e\rangle\ \{Q\}}$$

$$\text{"Ping"} = \text{"Ping"} * \exists \psi.\ sa \Mapsto \psi * (\forall m'.\ m'.\mathsf{str} = \text{"Pong"} \mathrel{-\!\!*} \psi\ m')$$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathtt{pong}} \Rrightarrow \Phi_{srv}\}$
  `let` $sh =$ `socket in`
$\{sh \xrightarrow{sa.\mathtt{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathtt{pong}} \Rrightarrow \Phi_{srv}\}$
  `socketbind` $sh\ sa$;
$\{sh \xrightarrow{sa.\mathtt{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Rrightarrow \Phi_{clt} * sa_{\mathtt{pong}} \Rrightarrow \Phi_{srv}\}$
  `send` sh "Ping" $sa_{\mathtt{pong}}$;
  `let` $m =$ `recv` sh `in`
  `assert` (`fst` $m =$ "Pong")
$\{\mathsf{True}\}$

$$(\forall m'.\ m'.\mathsf{str} = \text{“Pong”} \mathrel{-\!\!*} \Phi_{\mathtt{clt}}\ m')$$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$
  `let sh = socket in`
$\{sh \xrightarrow{sa.\mathsf{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$
  `socketbind sh sa;`
$\{sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Rrightarrow \Phi_{clt} * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$
  `send sh` “Ping” sa_{pong};
  `let m = recv sh in`
  `assert (fst m =` “Pong”`)`
$\{\mathsf{True}\}$

$$(\forall m'.\ m'.\mathsf{str} = \text{“Pong”} \twoheadrightarrow m'.\mathsf{str} = \text{“Pong”})$$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$
  `let` $sh =$ `socket in`
$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$
  `socketbind` $sh\ sa$;
$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Rrightarrow \Phi_{clt} * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$
  `send` sh "Ping" sa_{pong};
  `let` $m =$ `recv` sh `in`
  `assert` (`fst` $m =$ "Pong")
$\{\mathsf{True}\}$

HT-SEND
$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (R, T) * dst \Rrightarrow \Phi * ((sa, str, dst) \notin T \Rightarrow \Phi\ (sa, str, dst))\}$
  $\langle sa.\mathtt{ip};\ \mathtt{send}\ sh\ str\ dst \rangle$
$\{w.\ w = () * sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (R, T) \cup \{(sa, str, dst)\}\}$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Mapsto \Phi_{srv}\}$
$\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$
$\{sh \xhookrightarrow{sa.\mathsf{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Mapsto \Phi_{srv}\}$
$\quad \mathtt{socketbind}\ sh\ sa;$
$\{sh \xhookrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Mapsto \Phi_{clt} * sa_{\mathsf{pong}} \Mapsto \Phi_{srv}\}$
$\quad \mathtt{send}\ sh\ \text{“Ping”}\ sa_{\mathsf{pong}};$
$\{sh \xhookrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Ping”}, sa_{\mathsf{pong}})\}) * sa \Mapsto \Phi_{clt} * sa_{\mathsf{pong}} \Mapsto \Phi_{srv}\}$
$\quad \mathtt{let}\ m = \mathtt{recv}\ sh\ \mathtt{in}$
$\quad \mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})$
$\{\mathsf{True}\}$

HT-SEND
$\{sh \xhookrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (R, T) * dst \Mapsto \Phi * ((sa, str, dst) \notin T \Rightarrow \Phi\ (sa, str, dst))\}$
$\quad \langle sa.\mathtt{ip};\ \mathtt{send}\ sh\ str\ dst \rangle$
$\{w.\ w = () * sh \xhookrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (R, T) \cup \{(sa, str, dst)\}\}$

Verification of the ping pong client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$
  $\mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$
$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$
  $\mathtt{socketbind}\ sh\ sa;$
$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Rrightarrow \Phi_{clt} * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$
  $\mathtt{send}\ sh\ \text{“Ping”}\ sa_{\mathrm{pong}};$
$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Ping”}, sa_{\mathrm{pong}})\}) * sa \Rrightarrow \Phi_{clt} * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$
  $\mathtt{let}\ m = \mathtt{recv}\ sh\ \mathtt{in}$
  $\mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})$
$\{\mathsf{True}\}$

Verification of the ping pong client - Proof

$\{\text{FreeAdd}$
  `let` sh
$\{sh \xrightarrow{sa.\mathtt{ip}}$
  `socke`
$\{sh \xrightarrow{sa.\mathtt{ip}}$
  `send`
$\{sh \xrightarrow{sa.\mathtt{ip}} \text{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Ping”}, sa_{\text{pong}})\}) * sa \mapsto \Phi_{clt} * sa_{\text{pong}} \mapsto \Phi_{srv}\}$
  `let` $m =$ `recv` sh `in`
  `assert` (`fst` $m =$ “Pong”)
$\{\text{True}\}$

HT-RECV

$$\left\{sh \xrightarrow{sa.\mathtt{ip}} \text{Some}(sa) * sa \rightsquigarrow (R, T) * sa \mapsto \Phi\right\}$$
$$\langle sa.\mathtt{ip};\ \mathtt{recv}\ sh\rangle$$
$$\left\{\begin{array}{l} w.\ \exists str, src.\ w = (str, src) * sh \xrightarrow{sa.\mathtt{ip}} \text{Some}(sa) * \\ \quad sa \rightsquigarrow (R \cup \{(src, str, sa)\}, T) * \\ \quad ((src, str, sa) \notin R \Rightarrow \Phi\ (src, str, sa)) \end{array}\right\}$$

Verification of the ping pong client - Proof

$\{\text{FreeAdd}$

`let sh`

$\{sh \xrightarrow{sa.\mathtt{ip}}$

`socke`

$\{sh \xrightarrow{sa.\mathtt{ip}}$

`send`

HT-RECV

$$\{sh \xrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa) * sa \rightsquigarrow (R, T) * sa \Mapsto \Phi\}$$
$$\langle sa.\mathtt{ip};\ \mathtt{recv}\ sh\rangle$$
$$\left\{\begin{array}{l} w.\ \exists str, src.\ w = (str, src) * sh \xrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa) * \\ \quad sa \rightsquigarrow (R \cup \{(src, str, sa)\}, T) * \\ \quad ((src, str, sa) \notin R \Rightarrow \Phi\ (src, str, sa)) \end{array}\right\}$$

$\{sh \xrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Ping”}, sa_{\mathrm{pong}})\}) * sa \Mapsto \Phi_{clt} * sa_{\mathrm{pong}} \Mapsto \Phi_{srv}\}$

`let` $m =$ `recv` sh `in`

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa) * sa \rightsquigarrow (\{(src, str, sa)\}, \{(sa, \text{“Ping”}, sa_{\mathrm{pong}})\}) * \\ sa \Mapsto \Phi_{clt} * sa_{\mathrm{pong}} \Mapsto \Phi_{srv} * m = (str, src) * \Phi_{clt}\ (src, str, sa) \end{array}\right\}$$

`assert (fst` $m =$ “Pong”)

$\{\text{True}\}$

Verification of the ping pong client - Proof

$$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Mapsto \Phi_{srv}\}$$
$$\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$$
$$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Mapsto \Phi_{srv}\}$$
$$\quad \mathtt{socketbind}\ sh\ sa;$$
$$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Mapsto \Phi_{clt} * sa_{\mathrm{pong}} \Mapsto \Phi_{srv}\}$$
$$\quad \mathtt{send}\ sh\ \text{“Ping”}\ sa_{\mathrm{pong}};$$
$$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Ping”}, sa_{\mathrm{pong}})\}) * sa \Mapsto \Phi_{clt} * sa_{\mathrm{pong}} \Mapsto \Phi_{srv}\}$$
$$\quad \mathtt{let}\ m = \mathtt{recv}\ sh\ \mathtt{in}$$
$$\left\{\begin{array}{l} sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(src, str, sa)\}, \{(sa, \text{“Ping”}, sa_{\mathrm{pong}})\}) * \\ sa \Mapsto \Phi_{clt} * sa_{\mathrm{pong}} \Mapsto \Phi_{srv} * m = (str, src) * \Phi_{clt}\ (src, str, sa) \end{array}\right\}$$
$$\quad \mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})$$
$$\{\mathsf{True}\}$$

Verification of the ping pong client - Proof

$$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Mapsto \Phi_{srv}\}$$
$$\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$$
$$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Mapsto \Phi_{srv}\}$$
$$\quad \mathtt{socketbind}\ sh\ sa;$$
$$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Mapsto \Phi_{clt} * sa_{\mathrm{pong}} \Mapsto \Phi_{srv}\}$$
$$\quad \mathtt{send}\ sh\ \text{“Ping”}\ sa_{\mathrm{pong}};$$
$$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Ping”}, sa_{\mathrm{pong}})\}) * sa \Mapsto \Phi_{clt} * sa_{\mathrm{pong}} \Mapsto \Phi_{srv}\}$$
$$\quad \mathtt{let}\ m = \mathtt{recv}\ sh\ \mathtt{in}$$
$$\left\{\begin{array}{l} sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(src, str, sa)\}, \{(sa, \text{“Ping”}, sa_{\mathrm{pong}})\}) * \\ sa \Mapsto \Phi_{clt} * sa_{\mathrm{pong}} \Mapsto \Phi_{srv} * m = (str, src) * str = \text{“Pong”} \end{array}\right\}$$
$$\quad \mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})$$
$$\{\mathsf{True}\}$$

Verification of the echo example - Socket Protocols

```
clt_echo sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  let m_1 = recv sh in
  send sh "World" sa_echo;
  let m_2 = recvfresh sh [m_1] in
  assert (fst m_1 = "Hello");
  assert (fst m_2 = "World")
```

```
srv_echo ≜
  let sh = socket in
  socketbind sh sa_echo;
  rec go _ =
    (let m = recv sh in
     send sh (fst m) (snd m);
     go ()) ()
```

Verification of the echo example - Socket Protocols

```
clt_echo sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  let m1 = recv sh in
  send sh "World" sa_echo;
  let m2 = recvfresh sh [m1] in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

```
srv_echo ≜
  let sh = socket in
  socketbind sh sa_echo;
  rec go _ =
    (let m = recv sh in
     send sh (fst m) (snd m);
     go ()) ()
```

Socket protocols:

$\Phi_{clt} \triangleq \lambda m. \ldots$

$\Phi_{srv} \triangleq \lambda m. \ldots$

Verification of the echo example - Socket Protocols

```
clt_echo sa ≜
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  let m1 = recv sh in
  send sh "World" sa_echo;
  let m2 = recvfresh sh [m1] in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

```
srv_echo ≜
  let sh = socket in
  socketbind sh sa_echo;
  rec go _ =
    (let m = recv sh in
     send sh (fst m) (snd m);
     go ()) ()
```

Socket protocols:

$\Phi_{clt} \triangleq \lambda m.\ \ldots$

$\Phi_{srv} \triangleq \lambda m.\ \exists \psi.\ m.\mathsf{src} \Rightarrow \psi * \ldots$

Verification of the echo example - Socket Protocols

$\texttt{clt}_{\texttt{echo}}\ sa \triangleq$
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_{echo};
  let m_1 = recv sh in
  send sh "World" sa_{echo};
  let m_2 = recvfresh sh $[m_1]$ in
  assert (fst m_1 = "Hello");
  assert (fst m_2 = "World")

$\texttt{srv}_{\texttt{echo}} \triangleq$
  let sh = socket in
  socketbind sh sa_{echo};
  rec go _ =
    (let m = recv sh in
     send sh (fst m) (snd m);
     go ()) ()

Socket protocols:

$\Phi_{clt} \triangleq \lambda m.\ \ldots$

$\Phi_{srv} \triangleq \lambda m.\ \exists \psi.\ m.\mathsf{src} \Mapsto \psi\ *$
$(\forall m'.\ m'.\mathsf{str} = m.\mathsf{str} \mathrel{-\!\!*} \psi\ m')$

Verification of the echo example - Socket Protocols

$\texttt{clt}_{\texttt{echo}}\ sa \triangleq$
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_{echo};
  let m_1 = recv sh in
  send sh "World" sa_{echo};
  let m_2 = recvfresh sh $[m_1]$ in
  assert (fst m_1 = "Hello");
  assert (fst m_2 = "World")

$\texttt{srv}_{\texttt{echo}} \triangleq$
  let sh = socket in
  socketbind sh sa_{echo};
  rec go _ =
    (let m = recv sh in
     send sh (fst m) (snd m);
     go ()) ()

Socket protocols:

$\Phi_{clt} \triangleq \lambda m.\ ???$

$\Phi_{srv} \triangleq \lambda m.\ \exists \psi.\ m.\text{src} \Mapsto \psi\ *$
$(\forall m'.\ m'.\text{str} = m.\text{str} \mathrel{-\!\!*} \psi\ m')$

Verification of the echo example - Socket Protocols

User-defined ghost state!

$$t, u, P, Q ::= \ldots \mid \texttt{half}^{\gamma}\ x \mid \ldots$$

HALF-ALLOC

$$\vdash \Rrightarrow \exists \gamma.\, \texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ x$$

HALF-AGREE

$$\texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ y \vdash x = y$$

HALF-UPDATE

$$\texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ y \vdash \Rrightarrow \texttt{half}^{\gamma}\ z * \texttt{half}^{\gamma}\ z$$

Socket protocols:

$\Phi_{clt} \triangleq \lambda m.\ ???$

$\Phi_{srv} \triangleq \lambda m.\ \exists \psi.\ m.\mathsf{src} \Rrightarrow \psi * (\forall m'.\ m'.\mathsf{str} = m.\mathsf{str} \mathrel{-\!\!*} \psi\ m')$

Verification of the echo example - Socket Protocols

User-defined ghost state!

$$t, u, P, Q ::= \ldots \mid \mathtt{half}^{\gamma}\ x \mid \ldots$$

HALF-ALLOC

$$\vdash \Rrightarrow \exists \gamma.\, \mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ x$$

HALF-AGREE

$$\mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ y \vdash x = y$$

HALF-UPDATE

$$\mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ y \vdash \Rrightarrow \mathtt{half}^{\gamma}\ z * \mathtt{half}^{\gamma}\ z$$

Socket protocols:

$\Phi_{clt}\ \gamma \triangleq \lambda m.\, \mathtt{half}^{\gamma}\ m.\mathsf{str}$

$\Phi_{srv} \triangleq \lambda m.\, \exists \psi.\, m.\mathsf{src} \Rightarrow \psi *$
$(\forall m'.\, m'.\mathsf{str} = m.\mathsf{str} \mathrel{-\!\!*} \psi\ m')$

Verification of the echo example - Socket Protocols

User-defined ghost state!

$$t, u, P, Q ::= \ldots \mid \mathtt{half}^{\gamma}\ x \mid \ldots$$

HALF-ALLOC

$$\vdash \Rrightarrow \exists \gamma.\, \mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ x$$

HALF-AGREE

$$\mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ y \vdash x = y$$

HALF-UPDATE

$$\mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ y \vdash \Rrightarrow \mathtt{half}^{\gamma}\ z * \mathtt{half}^{\gamma}\ z$$

Socket protocols:

$\Phi_{clt}\ \gamma \triangleq \lambda m.\, \mathtt{half}^{\gamma}\ m.\mathsf{str}$

$\Phi_{srv} \triangleq \lambda m.\, \exists \psi.\, \exists \gamma.\, m.\mathsf{src} \Mapsto \psi *$
$(\forall m'.\, m'.\mathsf{str} = m.\mathsf{str} * \mathtt{half}^{\gamma}\ m.\mathsf{str} \mathrel{-\!\!*} \psi\ m')$

Verification of the echo example - Socket Protocols

User-defined ghost state!

$$t, u, P, Q ::= \ldots \mid \texttt{half}^{\gamma}\ x \mid \ldots$$

HALF-ALLOC

$$\frac{}{\vdash \Rrightarrow \exists \gamma.\, \texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ x}$$

HALF-AGREE

$$\frac{}{\texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ y \vdash x = y}$$

HALF-UPDATE

$$\frac{}{\texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ y \vdash \Rrightarrow \texttt{half}^{\gamma}\ z * \texttt{half}^{\gamma}\ z}$$

Socket protocols:

$\Phi_{clt}\ \gamma \triangleq \lambda m.\, \texttt{half}^{\gamma}\ m.\mathsf{str}$

$\Phi_{srv} \triangleq \lambda m.\, \exists \psi.\, \exists \gamma.\, \texttt{half}^{\gamma}\ m.\mathsf{str} * m.\mathsf{src} \Mapsto \psi *$
$(\forall m'.\, m'.\mathsf{str} = m.\mathsf{str} * \texttt{half}^{\gamma}\ m.\mathsf{str} \mathrel{-\!\!*} \psi\ m')$

Verification of the echo example - Socket Protocols

User-defined ghost state!

$$t, u, P, Q ::= \ldots \mid \mathtt{half}^{\gamma}\ x \mid \ldots$$

HALF-ALLOC

$$\vdash \Rightarrow\!\!\!\mid \exists\gamma.\, \mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ x$$

HALF-AGREE

$$\mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ y \vdash x = y$$

HALF-UPDATE

$$\mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ y \vdash \Rightarrow\!\!\!\mid \mathtt{half}^{\gamma}\ z * \mathtt{half}^{\gamma}\ z$$

Socket protocols:

$\Phi_{clt}\ \gamma \triangleq \lambda m.\, \mathtt{half}^{\gamma}\ m.\mathsf{str}$

$\Phi_{srv} \triangleq \lambda m.\, \exists\psi.\, \exists R.\, R * m.\mathsf{src} \Rightarrow\!\!\!\mid \psi * (\forall m'.\, m'.\mathsf{str} = m.\mathsf{str} * R \mathrel{-\!\!*} \psi\ m')$

Verification of the echo client - Proof

$$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\texttt{pong}} \Rrightarrow \Phi_{srv}\}$$
$$\langle sa.\texttt{ip}; \texttt{clt}_{\texttt{echo}}\ sa \rangle$$
$$\{\mathsf{True}\}$$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathtt{pong}} \Rightarrow \Phi_{srv}\}$

```
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  let m1 = recv sh in
  send sh "World" sa_echo;
  let m2 = recvfresh sh [m1] in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

$\{\mathsf{True}\}$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$

$\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\{sh \xhookrightarrow{sa.\mathsf{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$

$\quad \mathtt{socketbind}\ sh\ sa;$

$\quad \mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$\quad \mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$\quad \mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\quad \mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$\quad \mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\quad \mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$

$\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathrm{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$

$\quad \mathtt{socketbind}\ sh\ sa;$

$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathrm{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$

$\quad \mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$\quad \mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$\quad \mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\quad \mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$\quad \mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\quad \mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$

$\mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\{sh \xhookrightarrow{sa.\mathsf{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$

$\mathtt{socketbind}\ sh\ sa;$

$\{sh \xhookrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$

$\mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$\mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$\mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$\mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

HALF-ALLOC

$$\vdash \Rrightarrow \exists \gamma.\, \mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ x$$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$

$\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\{sh \xhookrightarrow{sa.\mathsf{ip}} \mathsf{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$

$\quad \mathtt{socketbind}\ sh\ sa;$

$\left\{ sh \xhookrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv} * \right.$
$\left. \mathtt{half}^{\gamma}\ \text{“Hello”} * \mathtt{half}^{\gamma}\ \text{“Hello”} \right\}$

$\quad \mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$\quad \mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$\quad \mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\quad \mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$\quad \mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\quad \mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

HALF-ALLOC

$$\vdash \Rrightarrow \exists \gamma.\, \mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ x$$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$

$\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathrm{None} * \mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$

$\quad \mathtt{socketbind}\ sh\ sa;$

$\left\{ \begin{array}{l} sh \xhookrightarrow{sa.\mathrm{ip}} \mathrm{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\ * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} * \mathtt{half}^{\gamma}\ \text{“Hello”} \end{array} \right\}$

$\quad \mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$\quad \mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$\quad \mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\quad \mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$\quad \mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\quad \mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$

$\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\quad \mathtt{socketbind}\ sh\ sa;$

$\left\{ sh \xhookrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\ * \right.$

$\left. \mathtt{half}^{\gamma}\ \text{“Hello”} * \mathtt{half}^{\gamma}\ \text{“Hello”} \right\}$

$\quad \mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$\quad \mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$\quad \mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\quad \mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$\quad \mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\quad \mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

Verification of the echo client - Proof

$$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$$
$$\mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$$
$$\mathtt{socketbind}\ sh\ sa;$$
$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathrm{ip}} \mathrm{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\ * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} * \mathtt{half}^{\gamma}\ \text{“Hello”} \end{array}\right\}$$
$$\mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$$
$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathrm{ip}} \mathrm{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\ * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} \end{array}\right\}$$
$$\mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$$
$$\mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$$
$$\mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$$
$$\mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$$
$$\mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$$
$$\{\mathsf{True}\}$$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$

$\mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\mathtt{socketbind}\ sh\ sa;$

$\mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\ * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} \end{array}\right\}$$

$\mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$\mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$\mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$

$\mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\mathtt{socketbind}\ sh\ sa;$

$\mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$$\left\{ \begin{array}{l} sh \xrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} \end{array} \right\}$$

$\mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$$\left\{ \begin{array}{l} sh \xrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(\_, str_1, sa)\}, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} * m_1 = (str_1, \_) * \Phi_{clt}\ \gamma\ (\_, str_1, sa) \end{array} \right\}$$

$\mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$\mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

Verification of the echo client - Proof

$$
\begin{array}{l}
\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\} \\
\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in} \\
\quad \mathtt{socketbind}\ sh\ sa; \\
\quad \mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo}; \\
\left\{ \begin{array}{l} sh \xrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} \end{array} \right\} \\
\quad \mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in} \\
\left\{ \begin{array}{l} sh \xrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(\_, str_1, sa)\}, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} * m_1 = (str_1, \_) * \mathtt{half}^{\gamma}\ str_1 \end{array} \right\} \\
\quad \mathtt{send}\ sh\ \text{“World”}\ sa_{echo}; \\
\quad \mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in} \\
\quad \mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”}); \\
\quad \mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”}) \\
\{\mathsf{True}\}
\end{array}
$$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$

$\mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\mathtt{socketbind}\ sh\ sa;$

$\mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} \end{array}\right\}$$

$\mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(\_, str_1, sa)\}, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} * m_1 = (str_1, \_) * \mathtt{half}^{\gamma}\ str_1 \end{array}\right\}$$

$\mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$\mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

HALF-AGREE

$$\mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ y \vdash x = y$$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$

$\mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\mathtt{socketbind}\ sh\ sa;$

$\mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} \end{array}\right\}$$

$\mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(\_, str_1, sa)\}, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} * m_1 = (\text{“Hello”}, \_) * \mathtt{half}^{\gamma}\ str_1 \end{array}\right\}$$

$\mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$\mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

HALF-AGREE

$$\frac{}{\mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ y \vdash x = y}$$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$

$\mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\mathtt{socketbind}\ sh\ sa;$

$\mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\ * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} \end{array}\right\}$$

$\mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(\_, str_1, sa)\}, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\ * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} * m_1 = (\text{“Hello”}, \_) * \mathtt{half}^{\gamma}\ str_1 \end{array}\right\}$$

$\mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$\mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

HALF-UPDATE

$$\frac{}{\mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ y \vdash \Rrightarrow \mathtt{half}^{\gamma}\ z * \mathtt{half}^{\gamma}\ z}$$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$

$\mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\mathtt{socketbind}\ sh\ sa;$

$\mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“Hello”} \end{array}\right\}$$

$\mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(\_, str_1, sa)\}, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“World”} * m_1 = (\text{“Hello”}, \_) * \mathtt{half}^{\gamma}\ \text{“World”} \end{array}\right\}$$

$\mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$\mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

HALF-UPDATE

$$\frac{}{\mathtt{half}^{\gamma}\ x * \mathtt{half}^{\gamma}\ y \vdash \Rrightarrow \mathtt{half}^{\gamma}\ z * \mathtt{half}^{\gamma}\ z}$$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$

$\mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\mathtt{socketbind}\ sh\ sa;$

$\mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$\mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$$\left\{ \begin{array}{l} sh \xrightarrow{sa.\mathrm{ip}} \mathrm{Some}(sa) * sa \rightsquigarrow (\{(\_, str_1, sa)\}, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“World”} * m_1 = (\text{“Hello”}, \_) * \mathtt{half}^{\gamma}\ \text{“World”} \end{array} \right\}$$

$\mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$\mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$

```
let sh = socket in
socketbind sh sa;
send sh "Hello" sa_echo;
let m1 = recv sh in
```

$$\left\{ \begin{array}{l} sh \xrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(\_, str_1, sa)\}, \{(sa, \text{“Hello”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“World”} * m_1 = (\text{“Hello”}, \_) * \mathtt{half}^{\gamma}\ \text{“World”} \end{array} \right\}$$

```
send sh "World" sa_echo;
```

$$\left\{ \begin{array}{l} sh \xrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(\_, str_1, sa)\}, \{(sa, \text{“Hello”}, sa_{echo}), (sa, \text{“World”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“World”} * m_1 = (\text{“Hello”}, \_) \end{array} \right\}$$

```
let m2 = recvfresh sh [m1] in
assert (fst m1 = "Hello");
assert (fst m2 = "World")
```

$\{\mathsf{True}\}$

Verification of the echo client - Proof

$$
\begin{array}{l}
\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\} \\
\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in} \\
\quad \mathtt{socketbind}\ sh\ sa; \\
\quad \mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo}; \\
\quad \mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in} \\
\quad \mathtt{send}\ sh\ \text{“World”}\ sa_{echo}; \\
\left\{ \begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(\_, str_1, sa)\}, \{(sa, \text{“Hello”}, sa_{echo}), (sa, \text{“World”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\ * \\ \mathtt{half}^{\gamma}\ \text{“World”} * m_1 = (\text{“Hello”}, \_) \end{array} \right\} \\
\quad \mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in} \\
\quad \mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”}); \\
\quad \mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”}) \\
\{\mathsf{True}\}
\end{array}
$$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$

$\mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$
$\mathtt{socketbind}\ sh\ sa;$
$\mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$
$\mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$
$\mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(\_, str_1, sa)\}, \{(sa, \text{“Hello”}, sa_{echo}), (sa, \text{“World”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“World”} * m_1 = (\text{“Hello”}, \_) \end{array}\right\}$$

$\mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\_, \_) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“World”} * m_1 = (\text{“Hello”}, \_) * m_2 = (str_2, \_) * \Phi_{clt}\ \gamma\ (\_, str_2, sa) \end{array}\right\}$$

$\mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$
$\mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

Verification of the echo client - Proof

$$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\}$$

$\mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$
$\mathtt{socketbind}\ sh\ sa;$
$\mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$
$\mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$
$\mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(\_, str_1, sa)\}, \{(sa, \text{“Hello”}, sa_{echo}), (sa, \text{“World”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“World”} * m_1 = (\text{“Hello”}, \_) \end{array}\right\}$$

$\mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\_, \_) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“World”} * m_1 = (\text{“Hello”}, \_) * m_2 = (str_2, \_) * \mathtt{half}^{\gamma}\ str_2 \end{array}\right\}$$

$\mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$
$\mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$$\{\mathsf{True}\}$$

Verification of the echo client - Proof

$$
\begin{array}{l}
\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv}\} \\
\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in} \\
\quad \mathtt{socketbind}\ sh\ sa; \\
\quad \mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo}; \\
\quad \mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in} \\
\quad \mathtt{send}\ sh\ \text{“World”}\ sa_{echo}; \\
\left\{ \begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\{(\_, str_1, sa)\}, \{(sa, \text{“Hello”}, sa_{echo}), (sa, \text{“World”}, sa_{echo})\}) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“World”} * m_1 = (\text{“Hello”}, \_) \end{array} \right\} \\
\quad \mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in} \\
\left\{ \begin{array}{l} sh \xrightarrow{sa.\mathsf{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\_, \_) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathsf{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“World”} * m_1 = (\text{“Hello”}, \_) * m_2 = (\text{“World”}, \_) * \mathtt{half}^{\gamma}\ str_2 \end{array} \right\} \\
\quad \mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”}); \\
\quad \mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”}) \\
\{\mathsf{True}\}
\end{array}
$$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv}\}$

$\mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in}$

$\mathtt{socketbind}\ sh\ sa;$

$\mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo};$

$\mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in}$

$\mathtt{send}\ sh\ \text{“World”}\ sa_{echo};$

$\mathtt{let}\ m_2 = \mathtt{recvfresh}\ sh\ [m_1]\ \mathtt{in}$

$$\left\{\begin{array}{l} sh \xrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\_, \_) * \\ sa \Rrightarrow \Phi_{clt}\ \gamma * sa_{\mathrm{pong}} \Rrightarrow \Phi_{srv} * \\ \mathtt{half}^{\gamma}\ \text{“World”} * m_1 = (\text{“Hello”}, \_) * m_2 = (\text{“World”}, \_) * \mathtt{half}^{\gamma}\ str_2 \end{array}\right\}$$

$\mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”});$

$\mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})$

$\{\mathsf{True}\}$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathtt{pong}} \Rightarrow \Phi_{srv}\}$

```
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  let m1 = recv sh in
  send sh "World" sa_echo;
  let m2 = recvfresh sh [m1] in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

$\{\mathsf{True}\}$

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathtt{pong}} \Rrightarrow \Phi_{srv}\}$

```
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  let m1 = recv sh in
  send sh "World" sa_echo;
  let m2 = recvfresh sh [m1] in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

$\{\mathsf{True}\}$

Is it safe?

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathtt{pong}} \Rrightarrow \Phi_{srv}\}$

```
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  let m1 = recv sh in
  send sh "World" sa_echo;
  let m2 = recvfresh sh [m1] in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

$\{\mathsf{True}\}$

Is it safe? Yes! ✓✓✓

Verification of the echo client - Proof

$\{\mathsf{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \mathtt{dyn}\ sa * sa_{\mathtt{pong}} \Rrightarrow \Phi_{srv}\}$

```
  let sh = socket in
  socketbind sh sa;
  send sh "Hello" sa_echo;
  let m1 = recv sh in
  send sh "World" sa_echo;
  let m2 = recvfresh sh [m1] in
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

$\{\mathsf{True}\}$

Is it safe? Yes! ✓✓✓

At least with respect to the distributed semantics.

Case Studies in Aneris

There are many case studies on top of Aneris

- Reliable Communication Library
- Distributed Causal Memory
- Conflict-Free Replicated Data Types

Case Studies in Aneris

There are many case studies on top of Aneris

- Reliable Communication Library
- Distributed Causal Memory
- Conflict-Free Replicated Data Types

There is also ongoing work happening

- Robust Safety
  - Safe network communication in the presence of malicious actors
- Progress Guarantees
  - Guarantees that the system does not wait indefinitely

Case Studies in Aneris

There are many case studies on top of Aneris

- Reliable Communication Library
- Distributed Causal Memory
- Conflict-Free Replicated Data Types

There is also ongoing work happening

- Robust Safety
  - Safe network communication in the presence of malicious actors
- Progress Guarantees
  - Guarantees that the system does not wait indefinitely

Feel free to
look around at https://github.com/logsem/aneris
and ask questions at hinrichsen@cs.au.dk