Aneris: Distributed Separation Logic

Jonas Kastberg Hinrichsen, Aarhus University

21. September 2022
Summer School on Security Testing and Verification

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

Hands-on presentation - Please ask questions!

Distributed Semantics

Distributed Semantics

A distributed semantics should consider:

1. The semantics of the individual nodes
   - e.g. node-local state changes, such as memory allocation

Distributed Semantics

A distributed semantics should consider:

1. The semantics of the individual nodes
   - e.g. node-local state changes, such as memory allocation
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
   - e.g. node-local state changes, such as memory allocation
2. The semantics of the network connectives
   - Socket allocation and binding
   - Message sending and receiving
3. The semantics of the (unreliable) network
   - Dropping, duplication, and reordering of messages

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:

$$v \in Val ::= () \mid b \mid i \mid s \mid \ell \mid \mathtt{rec}\ f\ x = e \mid \ldots$$

$$e \in Expr ::= v \mid x \mid \mathtt{rec}\ f\ x = e \mid e_1\ e_2 \mid \mathtt{ref}(e) \mid !\,e \mid e_1 \leftarrow e_2 \mid$$
$$\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \mid \mathtt{assert}\ e \mid \mathtt{fork}\,(e) \mid \ldots$$

Socket semantics inspired by UDP sockets:

$$v \in Val ::= \ \ldots \mid sh \mid sa \mid \ldots$$

$$e \in Expr ::= \ \ldots \mid \mathtt{socket} \mid \mathtt{socketbind}\ e_1\ e_2 \mid \mathtt{send}\ e_1\ e_2\ e_3 \mid \mathtt{recv}\ e \mid \ldots$$

AnerisLang: an OCaml-like language with UDP sockets

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

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

Socket semantics inspired by UDP sockets:

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

Network semantics are unreliable:

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

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) ()
```

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 ≜                              srv_pong ≜
  let sh = socket in                         let sh = socket in
  socketbind sh sa;                          socketbind sh sa_pong;
  send sh "Ping" sa_pong;                    rec go _ =
  let m = recv sh in                           (let m = recv sh in
  assert (fst m = "Pong")                       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".

$$
\begin{aligned}
&\texttt{clt}_{\text{pong}}\ sa \triangleq\\
&\quad \texttt{let}\ sh = \texttt{socket}\ \texttt{in}\\
&\quad \texttt{socketbind}\ sh\ sa;\\
&\quad \texttt{send}\ sh\ \text{“Ping”}\ sa_{\text{pong}};\\
&\quad \texttt{let}\ m = \texttt{recv}\ sh\ \texttt{in}\\
&\quad \texttt{assert}\ (\texttt{fst}\ m = \text{“Pong”})
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{srv}_{\text{pong}} \triangleq\\
&\quad \texttt{let}\ sh = \texttt{socket}\ \texttt{in}\\
&\quad \texttt{socketbind}\ sh\ sa_{\text{pong}};\\
&\quad \texttt{rec}\ go\ \_ =\\
&\qquad (\texttt{let}\ m = \texttt{recv}\ sh\ \texttt{in}\\
&\qquad\ \ \texttt{if}\ \texttt{fst}\ m = \text{“Ping”}\\
&\qquad\ \ \texttt{then}\ \texttt{send}\ sh\ \text{“Pong”}\ (\texttt{snd}\ m); go\ ()\\
&\qquad\ \ \texttt{else}\ \texttt{assert}\ \texttt{false})\ ()
\end{aligned}
$$

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".

$$\begin{array}{l}
\texttt{clt}_{\text{pong}}\ sa \triangleq \\
\quad \texttt{let}\ sh = \texttt{socket}\ \texttt{in} \\
\quad \texttt{socketbind}\ sh\ sa; \\
\quad \texttt{send}\ sh\ \text{“Ping”}\ sa_{\text{pong}}; \\
\quad \texttt{let}\ m = \texttt{recv}\ sh\ \texttt{in} \\
\quad \texttt{assert}\ (\texttt{fst}\ m = \text{“Pong”})
\end{array}$$

$$\begin{array}{l}
\texttt{srv}_{\text{pong}} \triangleq \\
\quad \texttt{let}\ sh = \texttt{socket}\ \texttt{in} \\
\quad \texttt{socketbind}\ sh\ sa_{\text{pong}}; \\
\quad \texttt{rec}\ go\ _ = \\
\qquad (\texttt{let}\ m = \texttt{recv}\ sh\ \texttt{in} \\
\qquad\ \texttt{if}\ \texttt{fst}\ m = \text{“Ping”} \\
\qquad\ \texttt{then}\ \texttt{send}\ sh\ \text{“Pong”}\ (\texttt{snd}\ m); go\ () \\
\qquad\ \texttt{else}\ \texttt{assert}\ \texttt{false})\ ()
\end{array}$$

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".

$$
\begin{array}{l}
\mathtt{clt}_{\mathrm{pong}}\ sa \triangleq \\
\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in} \\
\quad \mathtt{socketbind}\ sh\ sa; \\
\quad \mathtt{send}\ sh\ \text{“Ping”}\ sa_{\mathrm{pong}}; \\
\quad \mathtt{let}\ m = \mathtt{recv}\ sh\ \mathtt{in} \\
\quad \mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})
\end{array}
\qquad
\begin{array}{l}
\mathtt{srv}_{\mathrm{pong}} \triangleq \\
\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in} \\
\quad \mathtt{socketbind}\ sh\ sa_{\mathrm{pong}}; \\
\quad \mathtt{rec}\ go\ \_ = \\
\quad\quad (\mathtt{let}\ m = \mathtt{recv}\ sh\ \mathtt{in} \\
\quad\quad\ \mathtt{if}\ \mathtt{fst}\ m = \text{“Ping”} \\
\quad\quad\ \mathtt{then}\ \mathtt{send}\ sh\ \text{“Pong”}\ (\mathtt{snd}\ m); go\ () \\
\quad\quad\ \mathtt{else}\ \mathtt{assert}\ \mathtt{false})\ ()
\end{array}
$$

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 keep 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 keep 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".

```
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 ()) ()
```

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".

```
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?

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".

```
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?
- 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 ≜                          srv_echo ≜
  let sh = socket in                     let sh = socket in
  socketbind sh sa;                      socketbind sh sa_echo;
  send sh "Hello" sa_echo;               rec go _ =
  send sh "World" sa_echo;                 (let m = recv sh in
  let m1 = recv sh in                       send sh (fst m) (snd m);
  let m2 = recv sh in                       go ()) ()
  assert (fst m1 = "Hello");
  assert (fst m2 = "World")
```

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".

$$
\begin{array}{l}
\mathtt{clt_{echo}}\ sa \triangleq \\
\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in} \\
\quad \mathtt{socketbind}\ sh\ sa; \\
\quad \mathtt{send}\ sh\ \text{“Hello”}\ sa_{echo}; \\
\quad \mathtt{send}\ sh\ \text{“World”}\ sa_{echo}; \\
\quad \mathtt{let}\ m_1 = \mathtt{recv}\ sh\ \mathtt{in} \\
\quad \mathtt{let}\ m_2 = \mathtt{recv}\ sh\ \mathtt{in} \\
\quad \mathtt{assert}\ (\mathtt{fst}\ m_1 = \text{“Hello”}); \\
\quad \mathtt{assert}\ (\mathtt{fst}\ m_2 = \text{“World”})
\end{array}
\qquad
\begin{array}{l}
\mathtt{srv_{echo}} \triangleq \\
\quad \mathtt{let}\ sh = \mathtt{socket}\ \mathtt{in} \\
\quad \mathtt{socketbind}\ sh\ sa_{echo}; \\
\quad \mathtt{rec}\ go\ \_ = \\
\quad\quad (\mathtt{let}\ m = \mathtt{recv}\ sh\ \mathtt{in} \\
\quad\quad\ \ \mathtt{send}\ sh\ (\mathtt{fst}\ m)\ (\mathtt{snd}\ m); \\
\quad\quad\ \ go\ ())\ ()
\end{array}
$$

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
The client uses the service twice, first … rld”.

```
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”)
```

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

```
      in
      a_echo;

  (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 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 ()) ()
```

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 maintain **Abstraction** and **Modularity**:

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

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 maintain **Abstraction** and **Modularity**:

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

We want to guarantee **Safety**

- No node in the distributed system will ever get stuck

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:

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

Node-local rules for non-distributed expressions

Standard rules decorated with an ip identifier:

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

Example: rules for references:

HT-ALLOC

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

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

$$\tau, \sigma ::= \ldots \mid \mathit{Socket} \mid \mathit{Address} \mid \ldots$$

$$t, u, P, Q ::= \ldots \mid sh \overset{ip}{\hookrightarrow} o \mid \mathrm{FreeAddr}(sa) \mid \ldots$$

HT-NEWSOCKET

$$\{\mathrm{True}\}$$
$$\langle ip;\ \mathtt{socket}()\rangle$$
$$\{w.\ \exists sh.\ w = sh * sh \overset{ip}{\hookrightarrow} \mathrm{None}\}$$

HT-SOCKETBIND

$$\left\{sh \overset{sa.\mathtt{ip}}{\hookrightarrow} \mathrm{None} * \mathrm{FreeAddr}(sa)\right\}$$
$$\langle sa.\mathtt{ip};\ \mathtt{socketbind}\ sh\ sa\rangle$$
$$\{w.\ w = () * sh \overset{sa.\mathtt{ip}}{\hookrightarrow} \mathrm{Some}(sa)\}$$

Node-local rules for sockets

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

Ht-newsocket

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

Ht-socketbind

$$\left\{sh \xhookrightarrow{sa.\mathtt{ip}} \mathrm{None} * \mathrm{FreeAddr}(sa)\right\}$$
$$\langle sa.\mathtt{ip};\ \texttt{socketbind}\ sh\ sa\rangle$$
$$\{w.\ w = () * sh \xhookrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa)\}$$

Sockets are treated simlarly to references

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

Node-local rules for sockets

$$\tau, \sigma ::= \dots \mid \mathit{Socket} \mid \mathit{Address} \mid \dots$$
$$t, u, P, Q ::= \dots \mid sh \xhookrightarrow{ip} o \mid \mathrm{FreeAddr}(sa) \mid \dots$$

Ht-newsocket

$$\{\mathrm{True}\}\ \langle ip;\ \texttt{socket}()\rangle\ \{w.\, \exists sh.\, w = sh * sh \xhookrightarrow{ip} \mathrm{None}\}$$

Ht-socketbind

$$\{sh \xhookrightarrow{sa.\mathtt{ip}} \mathrm{None} * \mathrm{FreeAddr}(sa)\}\ \langle sa.\mathtt{ip};\ \texttt{socketbind}\ sh\ sa\rangle\ \{w.\, w = () * sh \xhookrightarrow{sa.\mathtt{ip}} \mathrm{Some}(sa)\}$$

Sockets are treated simlarly to references

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

All addresses are considered free on node startup

- i.e. the FreeAddr(sa) resource is obtained for any sa for free
- Guarantees that addresses are only bound once

Node-local rules for message passing

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

- Transfer 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 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 $m.\mathrm{src}$, $m.\mathrm{str}$, and $m.\mathrm{dst}$ for the message components

Node-local rules for message passing

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

- Transfer 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 $\Phi : \mathit{Message} \rightarrow \mathit{iProp}$ 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 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 $\Phi : \mathit{Message} \rightarrow \mathit{iProp}$ that all received messages must satisfy
  - Abstracts over reordering, as the order no longer matters
- Acquire resources specified 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 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

$$\tau, \sigma ::= \ \dots \mid \mathit{Message} \mid \dots$$

$$t, u, P, Q ::= \ \dots \mid sa \rightsquigarrow (R, T) \mid sa \Mapsto \Phi \mid \dots$$

Ht-send

$$\left\{\begin{array}{l} sh \xhookrightarrow{sa.\mathtt{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (R, T) * dst \Mapsto \Phi * \\ ((sa, str, dst) \notin T \Rightarrow \Phi\ (sa, str, dst)) \end{array}\right\}$$

$$\langle sa.\mathtt{ip};\ \mathtt{send}\ sh\ str\ dst \rangle$$

$$\left\{\begin{array}{l} w.\ w = () * sh \xhookrightarrow{sa.\mathtt{ip}} \mathsf{Some}(sa) * \\ \quad sa \rightsquigarrow (R, T \cup \{(sa, str, dst)\}) \end{array}\right\}$$

Ht-recv

$$\left\{ sh \xhookrightarrow{sa.\mathtt{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (R, T) * sa \Mapsto \Phi \right\}$$

$$\langle sa.\mathtt{ip};\ \mathtt{recv}\ sh \rangle$$

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

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

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

Ht-recv

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

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

$$\tau, \sigma ::= \ldots \mid \mathit{Message} \mid \ldots$$

$$t, u, P, Q ::= \ldots \mid sa \rightsquigarrow (R, T) \mid sa \Rrightarrow \Phi \mid \ldots$$

Ht-send

$$\left\{\begin{array}{l} sh \xhookrightarrow{sa.\mathtt{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (R, T) * dst \Rrightarrow \Phi * \\ ((sa, str, dst) \notin T \Rightarrow \Phi\ (sa, str, dst)) \end{array}\right\}$$

$$\langle sa.\mathtt{ip};\ \mathtt{send}\ sh\ str\ dst \rangle$$

$$\left\{\begin{array}{l} w.\ w = () * sh \xhookrightarrow{sa.\mathtt{ip}} \mathsf{Some}(sa) * \\ \quad sa \rightsquigarrow (R, T \cup \{(sa, str, dst)\}) \end{array}\right\}$$

Ht-recv

$$\{sh \xhookrightarrow{sa.\mathtt{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (R, T) * sa \Rrightarrow \Phi\}$$

$$\langle sa.\mathtt{ip};\ \mathtt{recv}\ sh \rangle$$

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

All addresses have empty histories on node startup

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

Protocols ($sa \Rrightarrow \Phi$) 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

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

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

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

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

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
- 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:

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

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

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:

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

$\Phi_{srv} \triangleq \lambda m.\ m.\mathrm{str} = \text{“Ping”} * \exists \psi.\ m.\mathrm{src} \Mapsto \psi * \ldots$

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:

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

$\Phi_{srv} \triangleq \lambda m.\ m.\mathsf{str} = \text{“Ping”} * \exists \psi.\ m.\mathsf{src} \Rrightarrow \psi * (\forall m'.\ m'.\mathsf{str} = \text{“Pong”} \twoheadrightarrow \psi\ m')$

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:

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

$\Phi_{srv} \triangleq \lambda m.\ m.\mathrm{str} = \text{“Ping”} * \exists \psi.\ m.\mathrm{src} \Mapsto \psi * (\forall m'.\ m'.\mathrm{str} = \text{“Pong”} \twoheadrightarrow \psi\ m')$

Verification of the ping pong client - Proof

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

Verification of the ping pong client - Proof

$\{\text{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\text{pong}} \Mapsto \Phi_{srv}\}$

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

$\{\text{True}\}$

Verification of the ping pong client - Proof

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

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

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

HT-NEWSOCKET

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

HT-SOCKETBIND

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

Verification of the ping pong client - Proof

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

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

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

$$\mathtt{socketbind}\ sh\ sa;$$
$$\mathtt{send}\ sh\ \text{“Ping”}\ sa_{\mathrm{pong}};$$
$$\mathtt{let}\ m = \mathtt{recv}\ sh\ \mathtt{in}$$
$$\mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})$$

$$\{\mathrm{True}\}$$

Ht-newsocket

$$\{\mathrm{True}\}$$
$$\langle ip;\ \mathtt{socket}()\rangle$$
$$\{w.\ \exists sh.\ w = sh * sh \xhookrightarrow{ip} \mathrm{None}\}$$

Ht-socketbind

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

Verification of the ping pong client - Proof

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

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

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

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

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

$\quad \mathtt{send}\ sh\ \text{“Ping”}\ sa_{\mathrm{pong}};$

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

$\quad \mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})$

$\{\mathrm{True}\}$

Ht-newsocket

$$\{\mathrm{True}\}\ \langle ip;\ \mathtt{socket}()\rangle\ \{w.\ \exists sh.\ w = sh * sh \xhookrightarrow{ip} \mathrm{None}\}$$

Ht-socketbind

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

Verification of the ping pong client - Proof

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

```
let sh = socket in
```

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

```
socketbind sh sa;
```

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

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

$\{\mathsf{True}\}$

Verification of the ping pong client - Proof

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

```
  let sh = socket in
```

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

```
  socketbind sh sa;
```

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

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

$\{\mathsf{True}\}$

Ht-send

$$\{sh \xhookrightarrow{sa.\mathtt{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (R, T) * dst \Mapsto \Phi * ((sa, str, dst) \notin T \Rightarrow \Phi\ (sa, str, dst))\}$$

$$\langle sa.\mathtt{ip};\ \mathtt{send}\ sh\ str\ dst\rangle$$

$$\{w.\ w = () * sh \xhookrightarrow{sa.\mathtt{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (R, T) \cup \{(sa, str, dst)\}\}$$

Verification of the ping pong client - Proof

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

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

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

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

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

$\mathtt{send}\ sh\ \text{“Ping”}\ sa_{\mathtt{pong}};$

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

$\mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})$

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

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

Verification of the ping pong client - Proof

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

```
let sh = socket in
```

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

```
socketbind sh sa;
```

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

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

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

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

Verification of the ping pong client - Proof

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

```
let sh = socket in
```

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

```
socketbind sh sa;
```

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

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

$$\{\text{True}\}$$

HT-DYNAMIC

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

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

Verification of the ping pong client - Proof

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

```
  let sh = socket in
```

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

```
  socketbind sh sa;
```

$\{sh \xhookrightarrow{sa.\text{ip}} \text{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Mapsto \Phi_{clt} * sa_{\text{pong}} \Mapsto \Phi_{srv}\}$

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

$\{\text{True}\}$

Ht-dynamic

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

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

Verification of the ping pong client - Proof

$\{\text{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\texttt{pong}} \Mapsto \Phi_{srv}\}$

```
  let sh = socket in
```

$\{sh \xhookrightarrow{sa.\texttt{ip}} \text{None} * \text{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\texttt{pong}} \Mapsto \Phi_{srv}\}$

```
  socketbind sh sa;
```

$\{sh \xhookrightarrow{sa.\texttt{ip}} \text{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Mapsto \Phi_{clt} * sa_{\texttt{pong}} \Mapsto \Phi_{srv}\}$

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

$\{\text{True}\}$

$$(\forall m'.\ m'.\text{str} = \text{"Pong"} \twoheadrightarrow \Phi_{\texttt{clt}}\ m')$$

Verification of the ping pong client - Proof

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

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

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

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

$$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Rightarrow \Phi_{clt} * sa_{\mathrm{pong}} \Rightarrow \Phi_{srv}\}$$

$\mathtt{send}\ sh\ \text{“Ping”}\ sa_{\mathrm{pong}};$

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

$\mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})$

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

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

Verification of the ping pong client - Proof

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

```
  let sh = socket in
```

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

```
  socketbind sh sa;
```

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

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

$\{\mathsf{True}\}$

HT-SEND

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

Verification of the ping pong client - Proof

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

```
let sh = socket in
```

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

```
socketbind sh sa;
```

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

```
send sh "Ping" sa_pong;
```

$\{sh \xhookrightarrow{sa.\text{ip}} \text{Some}(sa) * sa \rightsquigarrow (\emptyset, \{(sa, \text{"Ping"}, sa_{\text{pong}})\}) * sa \Rrightarrow \Phi_{clt} * sa_{\text{pong}} \Rrightarrow \Phi_{srv}\}$

```
let m = recv sh in
assert (fst m = "Pong")
```

$\{\text{True}\}$

Ht-send

$\{sh \xhookrightarrow{sa.\text{ip}} \text{Some}(sa) * sa \rightsquigarrow (R, T) * dst \Rrightarrow \Phi * ((sa, str, dst) \notin T \Rightarrow \Phi\ (sa, str, dst))\}$

$\langle sa.\text{ip};\ \texttt{send}\ sh\ str\ dst \rangle$

$\{w.\ w = () * sh \xhookrightarrow{sa.\text{ip}} \text{Some}(sa) * sa \rightsquigarrow (R, T) \cup \{(sa, str, dst)\}\}$

Verification of the ping pong client - Proof

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

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

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

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

$\{sh \xhookrightarrow{sa.\mathrm{ip}} \mathsf{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * sa \Mapsto \Phi_{clt} * sa_{\mathrm{pong}} \Mapsto \Phi_{srv}\}$

$\quad \mathtt{send}\ sh\ \text{“Ping”}\ sa_{\mathrm{pong}};$

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

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

$\quad \mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})$

$\{\mathsf{True}\}$

Verification of the ping pong client - Proof

$\{\text{FreeAdd}$

  let sh

$\{sh \xhookrightarrow{sa.\text{ip}}$

  socke

$\{sh \xhookrightarrow{sa.\text{ip}}$

  send

HT-RECV

$$\{sh \xhookrightarrow{sa.\text{ip}} \text{Some}(sa) * sa \rightsquigarrow (R, T) * sa \Rrightarrow \Phi\}$$

$$\langle sa.\text{ip};\ \texttt{recv}\ sh\rangle$$

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

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

```
  let m = recv sh in
  assert (fst m = "Pong")
```

$\{\text{True}\}$

Verification of the ping pong client - Proof

{FreeAdd

let *sh*

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

socke

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

send

HT-RECV

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

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

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

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

$\mathtt{assert}\ (\mathtt{fst}\ m = \text{“Pong”})$

{True}

Verification of the ping pong client - Proof

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

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

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

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

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

$$\quad \texttt{send}\ sh\ \text{``Ping''}\ sa_{\mathtt{pong}};$$

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

$$\quad \texttt{let}\ m = \texttt{recv}\ sh\ \texttt{in}$$

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

$$\quad \texttt{assert}\ (\texttt{fst}\ m = \text{``Pong''})$$

$$\{\mathrm{True}\}$$

Verification of the ping pong client - Proof

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

```
let sh = socket in
```

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

```
socketbind sh sa;
```

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

$\mathtt{send}\ sh$ "Ping" sa_{pong};

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

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

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

$\mathtt{assert}\ (\mathtt{fst}\ m =$ "Pong")

$$\{\text{True}\}$$

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 ()) ()
```

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:

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

$\Phi_{srv} \triangleq \lambda m. \exists \psi. \ m.\mathrm{src} \Rrightarrow \psi * \ \ldots$

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:

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

$\Phi_{srv} \triangleq \lambda m. \exists \psi. \ m.\mathrm{src} \Rrightarrow \psi \ *$
$\quad (\forall m'. \ m'.\mathrm{str} = m.\mathrm{str} \twoheadrightarrow \psi \ m')$

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:

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

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

Verification of the echo example - Socket Protocols

User-defined resources!

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

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:

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

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

Verification of the echo example - Socket Protocols

User-defined resources!

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

$$\frac{\textsc{Half-alloc}}{\vdash\ \Rrightarrow \exists \gamma.\, \texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ x} \qquad \frac{\textsc{Half-agree}}{\texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ y \vdash x = y}$$

$$\frac{\textsc{Half-update}}{\texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ y \vdash\ \Rrightarrow \texttt{half}^{\gamma}\ z * \texttt{half}^{\gamma}\ z}$$

Socket protocols:

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

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

Verification of the echo example - Socket Protocols

User-defined resources!

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

$$\frac{\textsc{Half-alloc}}{\vdash \Rrightarrow \exists \gamma. \texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ x}$$

$$\frac{\textsc{Half-agree}}{\texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ y \vdash x = y}$$

$$\frac{\textsc{Half-update}}{\texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ y \vdash \Rrightarrow \texttt{half}^{\gamma}\ z * \texttt{half}^{\gamma}\ z}$$

Socket protocols:

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

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

Verification of the echo example - Socket Protocols

User-defined resources!

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

$$\textsc{Half-alloc}\quad \frac{}{\vdash\ \Rrightarrow \exists \gamma.\, \texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ x}
\qquad
\textsc{Half-agree}\quad \frac{}{\texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ y \vdash x = y}$$

$$\textsc{Half-update}\quad \frac{}{\texttt{half}^{\gamma}\ x * \texttt{half}^{\gamma}\ y \vdash\ \Rrightarrow \texttt{half}^{\gamma}\ z * \texttt{half}^{\gamma}\ z}$$

Socket protocols:

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

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

Verification of the echo client - Proof

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

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

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

```
  let sh = socket in
```

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

```
  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

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

```
  let sh = socket in
```

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

```
  socketbind sh sa;
```

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

```
  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

$\{\text{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\texttt{pong}} \Mapsto \Phi_{srv}\}$

```
  let sh = socket in
```

$\{sh \xhookrightarrow{sa.\texttt{ip}} \text{None} * \text{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\texttt{pong}} \Mapsto \Phi_{srv}\}$

```
  socketbind sh sa;
```

$\{sh \xhookrightarrow{sa.\texttt{ip}} \text{Some}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\texttt{pong}} \Mapsto \Phi_{srv}\}$

```
  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")
```

$\{\text{True}\}$

HALF-ALLOC

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

Verification of the echo client - Proof

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

```
  let sh = socket in
```

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

```
  socketbind sh sa;
```

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

```
  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")
```

$\{\mathsf{True}\}$

Half-alloc

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

Verification of the echo client - Proof

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

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

$\{sh \xhookrightarrow{sa.\texttt{ip}} \text{None} * \text{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\texttt{pong}} \Rightarrow \Phi_{srv}\}$

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

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

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

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

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

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

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

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

$\{\text{True}\}$

Verification of the echo client - Proof

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

```
let sh = socket in
socketbind sh sa;
```

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

```
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")
```

$\{\text{True}\}$

Verification of the echo client - Proof

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

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

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

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

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

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

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

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

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

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

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

$$\{\mathrm{True}\}$$

Verification of the echo client - Proof

$\{\text{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\texttt{pong}} \Mapsto \Phi_{srv}\}$

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

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

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

$\{\text{True}\}$

Verification of the echo client - Proof

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

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

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

```
let m1 = recv sh in
```

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

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

$\{\text{True}\}$

Verification of the echo client - Proof

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

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

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

```
let m1 = recv sh in
```

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

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

$\{\text{True}\}$

Verification of the echo client - Proof

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

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

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

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

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

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

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

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

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

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

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

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

HALF-AGREE

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

Verification of the echo client - Proof

$\{\text{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\texttt{pong}} \Mapsto \Phi_{srv}\}$

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

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

```
let m1 = recv sh in
```

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

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

$\{\text{True}\}$

Half-agree

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

Verification of the echo client - Proof

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

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

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

```
let m1 = recv sh in
```

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

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

$\{\mathrm{True}\}$

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

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

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

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

```
let m1 = recv sh in
```

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

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

$\{\text{True}\}$

Half-update

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

Verification of the echo client - Proof

$$\{\text{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\texttt{pong}} \Mapsto \Phi_{srv}\}$$

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

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

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

$$\{\text{True}\}$$

Verification of the echo client - Proof

$\{\text{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\texttt{pong}} \Mapsto \Phi_{srv}\}$

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

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

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

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

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

$\{\text{True}\}$

Verification of the echo client - Proof

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

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

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

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

$\{\mathrm{True}\}$

Verification of the echo client - Proof

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

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

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

```
let m2 = recvfresh sh [m1] in
```

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

```
assert (fst m1 = "Hello");
assert (fst m2 = "World")
```

$\{\mathrm{True}\}$

Verification of the echo client - Proof

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

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

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

```
let m2 = recvfresh sh [m1] in
```

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

```
assert (fst m1 = "Hello");
assert (fst m2 = "World")
```

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

Verification of the echo client - Proof

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

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

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

```
let m2 = recvfresh sh [m1] in
```

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

```
assert (fst m1 = "Hello");
assert (fst m2 = "World")
```

$$\{\mathrm{True}\}$$

Verification of the echo client - Proof

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

```
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
```

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

```
assert (fst m1 = "Hello");
assert (fst m2 = "World")
```

$\{\mathrm{True}\}$

Verification of the echo client - Proof

$\{\text{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\text{pong}} \Mapsto \Phi_{srv}\}$

```
  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")
```

$\{\text{True}\}$

Verification of the echo client - Proof

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

```
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")
```

$\{\text{True}\}$

Is it safe?

Verification of the echo client - Proof

$\{\text{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\texttt{pong}} \Mapsto \Phi_{srv}\}$

```
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")
```

$\{\text{True}\}$

Is it safe? Yes! ✓✓✓

Verification of the echo client - Proof

$\{\text{FreeAddr}(sa) * sa \rightsquigarrow (\emptyset, \emptyset) * \texttt{dyn}\ sa * sa_{\texttt{pong}} \Mapsto \Phi_{srv}\}$

```
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")
```

$\{\text{True}\}$

Is it safe? Yes! ✓✓✓

At least with respect to the operational 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