Actris: Session-Type Based Reasoning in Separation Logic

Jonas Kastberg Hinrichsen
Jesper Bengtson
Robbert Krebbers

Use Cases

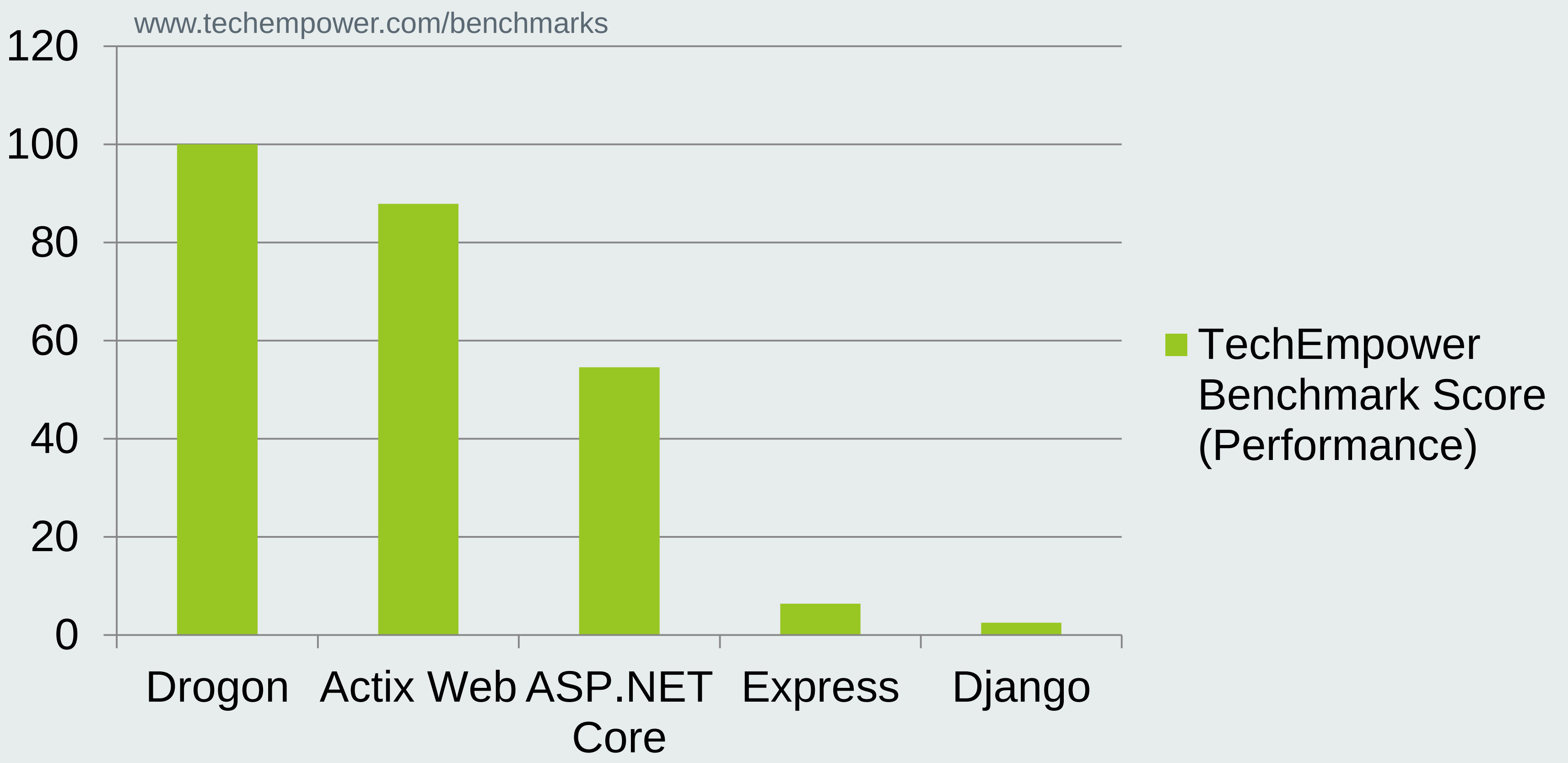
Message Passing in Applications

«From **network protocols** over the Internet to **server-client systems** in local area networks to **distributed applications** in the world wide web to **interaction between mobile robots** to a **global banking system**, [...]»

Honda et al. [1998]

Use Cases

Web Frameworks



Use Cases

Interprocess Communication

fabian@thinkpad-t470: ~/Dokumente

```
fabian@thinkpad-t470:~/Dokumente$ pdftotext faust.pdf - | grep -i faust | wc -l
272
fabian@thinkpad-t470:~/Dokumente$ pdftotext faust.pdf - | grep -i mephisto | wc -l
284
fabian@thinkpad-t470:~/Dokumente$ pdftotext faust.pdf - | grep -i margarete | wc -l
85
fabian@thinkpad-t470:~/Dokumente$
```

Problem: Mixed Concurrency Models

Example: Websocket Chat

```
type Users = Arc<RwLock<HashMap<usize,
mpsc::UnboundedSender<Result<Message, warp::Error>>>>>;

async fn user_connected(ws: WebSocket, users: Users) {
    let my_id = NEXT_USER_ID.fetch_add(1,
Ordering::Relaxed);

    let (tx, mut rx) = ws.split();

    users.write().await.insert(my_id, tx);

    // ...
}

async fn user_message(my_id: usize, msg: Message, users:
&Users) {
   for (&uid, tx) in users.read().await.iter() {
        if my_id != uid {
            if let Err(_disconnected) =
tx.send(Ok(Message::text(msg.unwrap().clone()))) {}
        }
    }
}

async fn user_disconnected(my_id: usize, users: &Users) {
    users.write().await.remove(&my_id);
}
```

github.com/seanmonstar/warp/blob/master/examples/websockets_chat.rs

Problem: Mixed Concurrency Models

Example: Websocket Chat

- Message Passing

```
type Users = Arc<RwLock<HashMap<usize,
mpsc::UnboundedSender<Result<Message, warp::Error>>>>>;

async fn user_connected(ws: WebSocket, users: Users) {
    let my_id = NEXT_USER_ID.fetch_add(1,
Ordering::Relaxed);

    let (tx, mut rx) = ws.split();

    users.write().await.insert(my_id, tx);

    // ...
}

async fn user_message(my_id: usize, msg: Message, users:
&Users) {
   for (&uid, tx) in users.read().await.iter() {
        if my_id != uid {
            if let Err(_disconnected) =
tx.send(Ok(Message::text(msg.unwrap().clone()))) {}
        }
    }
}

async fn user_disconnected(my_id: usize, users: &Users) {
    users.write().await.remove(&my_id);
}
```

github.com/seanmonstar/warp/blob/master/examples/websockets_chat.rs

Problem: Mixed Concurrency Models

Example: Websocket Chat

- Message Passing
- Readers-Writer Lock

```
type Users = Arc<RwLock<HashMap<usize,
mpsc::UnboundedSender<Result<Message, warp::Error>>>>>;

async fn user_connected(ws: WebSocket, users: Users) {
    let my_id = NEXT_USER_ID.fetch_add(1,
Ordering::Relaxed);

    let (tx, mut rx) = ws.split();

    users.write().await.insert(my_id, tx);

    // ...
}

async fn user_message(my_id: usize, msg: Message, users:
&Users) {
   for (&uid, tx) in users.read().await.iter() {
        if my_id != uid {
            if let Err(_disconnected) =
tx.send(Ok(Message::text(msg.unwrap().clone()))) {}
        }
    }
}

async fn user_disconnected(my_id: usize, users: &Users) {
    users.write().await.remove(&my_id);
}
```

github.com/seanmonstar/warp/blob/master/examples/websockets_chat.rs

Problem: Mixed Concurrency Models

Example: Websocket Chat

- Message Passing
- Readers-Writer Lock
- Atomic Types

```
type Users = Arc<RwLock<HashMap<usize,
mpsc::UnboundedSender<Result<Message, warp::Error>>>>>;

async fn user_connected(ws: WebSocket, users: Users) {
    let my_id = NEXT_USER_ID.fetch_add(1,
Ordering::Relaxed);

    let (tx, mut rx) = ws.split();

    users.write().await.insert(my_id, tx);

    // ...
}

async fn user_message(my_id: usize, msg: Message, users:
&Users) {
   for (&uid, tx) in users.read().await.iter() {
        if my_id != uid {
            if let Err(_disconnected) =
tx.send(Ok(Message::text(msg.unwrap().clone()))) {}
        }
    }
}

async fn user_disconnected(my_id: usize, users: &Users) {
    users.write().await.remove(&my_id);
}
```

github.com/seanmonstar/warp/blob/master/examples/websockets_chat.rs

Problem: Mixed Concurrency Models

Example: Websocket Chat

- Message Passing
- Readers-Writer Lock
- Atomic Types

Difficult to prove functional correctness of programs using mixed concurrency protocols

```
type Users = Arc<RwLock<HashMap<usize,
mpsc::UnboundedSender<Result<Message, warp::Error>>>>>;

async fn user_connected(ws: WebSocket, users: Users) {
    let my_id = NEXT_USER_ID.fetch_add(1,
Ordering::Relaxed);

    let (tx, mut rx) = ws.split();

    users.write().await.insert(my_id, tx);

    // ...
}

async fn user_message(my_id: usize, msg: Message, users:
&Users) {
   for (&uid, tx) in users.read().await.iter() {
        if my_id != uid {
            if let Err(_disconnected) =
tx.send(Ok(Message::text(msg.unwrap().clone()))) {}
        }
    }
}

async fn user_disconnected(my_id: usize, users: &Users) {
    users.write().await.remove(&my_id);
}
```

github.com/seanmonstar/warp/blob/master/examples/websockets_chat.rs

Goals

Create a logic for **proving functional correctness** of programs that use a **combination of message passing and other concurrency paradigms**

Hoare Logic

- Reason about the correctness of programs
- Describe how the state of a program changes

~~Hoare~~ Separation Logic

- Reason about the correctness of programs
- Describe how the state of a program changes

+ Ownership of resources

~~Hoare~~ Logic

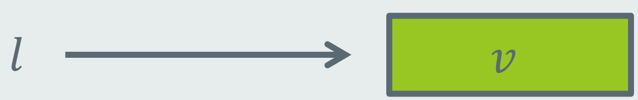
Separation

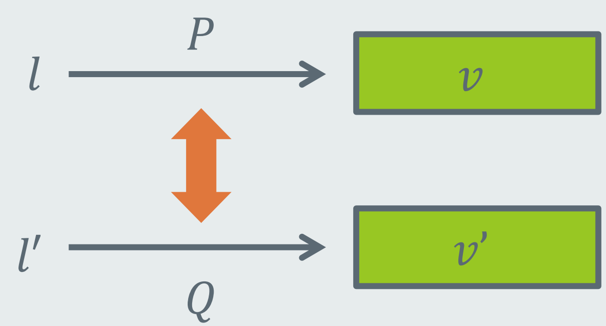
$$l \mapsto v$$

- Unique ownership of location l in store that points to value v in heap

$$P * Q$$

- Heap can be split into disjoint parts where P and Q hold respectively





Previous Work

Let's try to verify this program using previous state of the art.

$\{True\}$

```
let (c, c') = new_chan () in
fork {
    let l = recv c' in
    l <- !l + 2;
    send c' ()
}
let l = ref 40 in
send c l;
recv c;
!l
```

$\{42.\}$

Previous Work

Previous work (Bocchi et al. [2010]) cannot verify this because it does not support mutable state.

$\{True\}$

```
let (c, c') = new_chan () in
fork {
    let l = recv c' in
    l <- !l + 2;
    send c' ()
}
let l = ref 40 in
send c l;
recv c;
!l
```

$\{42.\}$

MOTIVATION

BACKGROUND

WHAT'S NEW

CASE STUDY

Previous Work

Previous work (Craciun et al. [2015]) cannot verify this because it does not support dependent protocols.

$\{True\}$

```
let (c, c') = new_chan () in
fork {
    let l = recv c' in
    l <- !l + 2;
    send c' ()
}
let l = ref 40 in
send c l;
recv c;
!l
```

$\{42.\}$

Previous Work

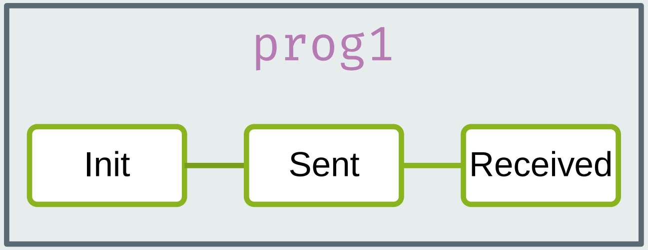
But with Actris, verification is possible, as we will see later!

$\{True\}$

```
let (c, c') = new_chan () in
fork {
    let l = recv c' in
    l <- !l + 2;
    send c' ()
}
let l = ref 40 in
send c l;
recv c;
!l
```

$\{42.\}$

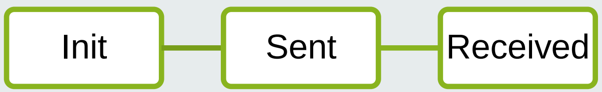
State Transition Systems



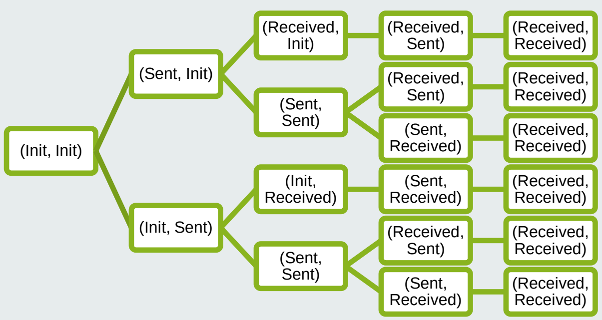

```
prog1:
    {True}
    let (c, c') = new_chan() in
    fork {
        send c' 42
    };
    recv c
    {42.}
```

State Transition Systems

`prog1`



`prog1 || prog1`

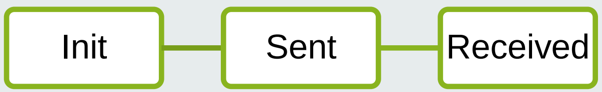


- Consider all possible interleavings

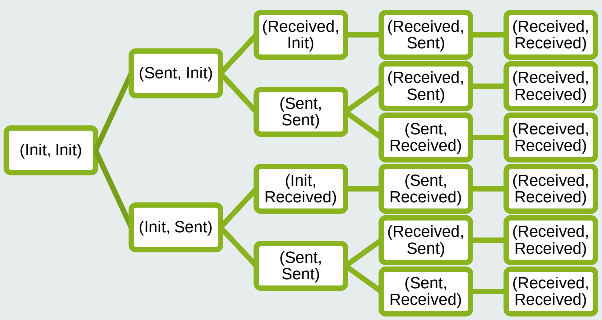
```
prog1:
    {True}
    let (c, c') = new_chan() in
    fork {
        send c' 42
    };
    recv c
    {42.}
```

State Transition Systems

prog1



prog1 || prog1



- Consider all possible interleavings

prog2

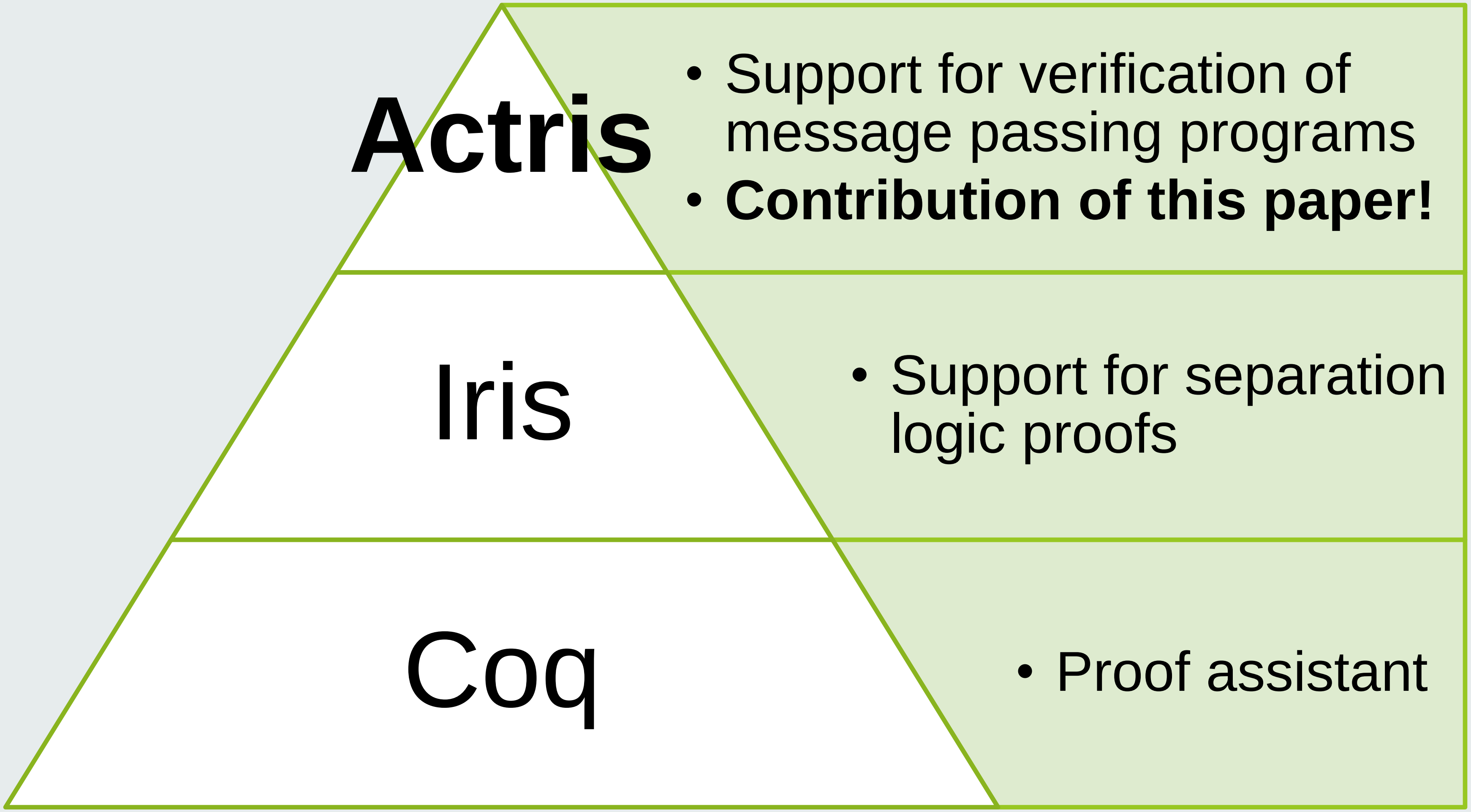
- Impossible to verify because we cannot send functions

```
prog1:
    {True}
    let (c, c') = new_chan() in
    fork {
        send c' 42
    };
    recv c
    {42.}
prog2:
    {True}
    let (c, c') = new_chan() in
    fork {
        let f = fun x -> x + 2 in
        send c' f
    };
    let f = recv c in
    f 40
    {42.}
```

Actris

- Extension of Iris separation logic to support message passing
  - Introduces **dependent separation protocols**
- ... but without some shortcomings of earlier work
  - Allows verification of multiple concurrency protocols
    - Programs using locks
  - Applicable on a wider range of programs
    - Recursion
    - Sending functions/channels over channels
    - Mutable/Shared state

Ecosystem


Actris
• Support for verification of message passing programs
• **Contribution of this paper!**
Iris
• Support for separation logic proofs
Coq
• Proof assistant

Dependent Separation Protocols

value being sent or received

send (!) or receive (?)

dependent separation protocol tail

$$c \rightarrowtail \,!\, \vec{x}{:}\vec{\tau}\langle v\rangle\{P\}.\,prot$$

unique ownership of endpoint c, and c follows this protocol

separation logic proposition possibly using v

binds x into v, P and prot

Dependent Separation Protocols

Send Type

$$c \rightarrowtail\ !\,(a\colon \mathbb{N})\langle a\rangle$$

«send integer value»

Dependent Separation Protocols

Duals

$$c \rightarrowtail\ !\,(a\!: \mathbb{N})\langle a \rangle$$

«send integer value»

$$c' \rightarrowtail \overline{!\,(a\!: \mathbb{N})\langle a \rangle} \equiv\ ?\,(a\!: \mathbb{N})\langle a \rangle$$

«receive integer value»

Dependent Separation Protocols

Composition

$$c \rightarrowtail \,!\,(a\colon \mathbb{N})\langle a\rangle \;?\,(a\colon \mathbb{N})\langle a\rangle$$

«send integer value,

then receive another integer value»

Dependent Separation Protocols

Propositions

$$c \rightarrowtail\ !\,(a\colon \mathbb{N})\langle a\rangle\{a > 10\}$$

«send integer value greater than 10»

Dependent Separation Protocols

References

$$c \rightarrowtail \,!\,(l\colon \mathrm{Loc})(b\colon \mathbb{N})\langle l\rangle\{l \mapsto b\}$$

«send reference to integer value»

Dependent Separation Protocols

Functions

$$c \rightarrowtail\ !\,(f\colon \mathbb{N} \to \mathbb{N} \to \mathbb{B})\langle f \rangle$$
$$\{\forall x, y \in \mathbb{N}\colon x \leq y \Rightarrow f(x) \leq f(y)\}$$

«send monotonically increasing function»

MOTIVATION
BACKGROUND
WHAT'S NEW
CASE STUDY

Dependent Separation Protocols

Delegation

$$c \rightarrowtail\ !\, c'\langle c'\rangle\{c' \rightarrowtail ?\,(a\colon \mathbb{N})\langle a\rangle\}$$

«send channel which receives integer value»

Message Passing Rules

$$\{\mathsf{True}\}\ \texttt{new\_chan}\ ()\ \{(c, c').\ c \rightarrowtail \mathit{prot} * c' \rightarrowtail \overline{\mathit{prot}}\} \qquad \text{(HT-NEWCHAN)}$$

- Duality guarantees that any receive is matched with a send, and the other way around
- c and c' can be separated

Message Passing Rules

$$\{\mathsf{True}\}\ \mathsf{new\_chan}\ ()\ \{(c, c').\ c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\} \qquad \text{(HT-NEWCHAN)}$$

- Duality guarantees that any receive is matched with a send, and the other way around
- c and c' can be separated

$$\{c \rightarrowtail !\,\vec{x}:\vec{\tau}\,\langle v\rangle\{P\}.\,prot * P[\vec{t}/\vec{x}]\}\ \mathsf{send}\ c\ (v[\vec{t}/\vec{x}])\ \{c \rightarrowtail prot[\vec{t}/\vec{x}]\} \qquad \text{(HT-SEND)}$$

- Requires send (!) type
- P holds in precondition, but doesn't hold in postcondition
- Give up ownership of resources in P

Message Passing Rules

$$\{\mathsf{True}\}\ \texttt{new\_chan}\ ()\ \{(c, c').\ c \rightarrowtail \mathit{prot} * c' \rightarrowtail \overline{\mathit{prot}}\} \qquad (\textsc{Ht-newchan})$$

- Duality guarantees that any receive is matched with a send, and the other way around
- c and c' can be separated

$$\{c \rightarrowtail !\, \vec{x} : \vec{\tau}\ \langle v \rangle \{P\}.\ \mathit{prot} * P[\vec{t}/\vec{x}]\}\ \texttt{send}\ c\ (v[\vec{t}/\vec{x}])\ \{c \rightarrowtail \mathit{prot}[\vec{t}/\vec{x}]\} \qquad (\textsc{Ht-send})$$

- Requires send (!) type
- P holds in precondition, but doesn't hold in postcondition
- Give up ownership of resources in P

$$\{c \rightarrowtail ?\, \vec{x} : \vec{\tau}\ \langle v \rangle \{P\}.\ \mathit{prot}\}\ \texttt{recv}\ c\ \{w.\ \exists \vec{y}.\ (w = v[\vec{y}/\vec{x}]) * c \rightarrowtail \mathit{prot}[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]\} \qquad (\textsc{Ht-recv})$$

- Requires receive (?) type
- P doesn't hold in precondition, but holds in postcondition
- Acqires the resources P

Dependent Separation Protocols

Let's try to verify this program!

$\{True\}$

```
let (c, c') = new_chan () in

fork {


    let l = recv c' in

    l <- !l + 2;


    send c' ()

}

let l = ref 40 in


send c l;


recv c;

!l
```

$\{42.\}$

Dependent Separation Protocols

$\{True\}$
```
let (c, c') = new_chan () in
```
$\{c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\}$
```
fork {
    let l = recv c' in
    l <- !l + 2;
    send c' ()
}
let l = ref 40 in
send c l;
recv c;
```

$$\{\mathrm{True}\}\ \mathsf{new\_chan}\ ()\ \{(c, c').\, c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\} \qquad \text{(HT-NEWCHAN)}$$

$\{42.\}$

Dependent Separation Protocols

$\{True\}$

```
let (c, c’) = new_chan () in
```

$\{c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\}$

```
fork {
```

$\{c' \rightarrowtail \overline{prot}\}$

```
    let l = recv c’ in

    l <- !l + 2;


    send c’ ()

}
```

$\{c \rightarrowtail prot\}$

```
let l = ref 40 in


send c l;


recv c;

!l
```

$\{42.\}$

Dependent Separation Protocols

$\{True\}$

```
let (c, c') = new_chan () in
```

$\{c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\}$

```
fork {
```

$\{c' \rightarrowtail \overline{prot}\}$

```
    let l = recv c' in

    l <- !l + 2;


    send c' ()

}
```

$\{c \rightarrowtail prot\}$

```
let l = ref 40 in
```

$\{c \rightarrowtail prot * l \mapsto x * x = 40\}$

```
send c l;


recv c;

!l
```

$\{42.\}$

Dependent Separation Protocols

$prot = {!}\, l\, x\langle l\rangle\, \{l \mapsto x\}. prot'$

$\{True\}$
```
let (c, c') = new_chan () in
```
$\{c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\}$
```
fork {
```
$\{c' \rightarrowtail \overline{prot}\} \vDash$

$\{c' \rightarrowtail ?\, l\, x\langle l\rangle\{l \mapsto x\}. \overline{prot'}\}$
```
     let l = recv c' in

     l <- !l + 2;


     send c' ()

}
```
$\{c \rightarrowtail prot\}$
```
let l = ref 40 in
```
$\{c \rightarrowtail prot * l \mapsto x * x = 40\} \vDash$

$\{c \rightarrowtail {!}\, l\, x\langle l\rangle\{l \mapsto x\}. prot' * l \mapsto x * x = 40\}$
```
send c l;


recv c;

!l
```
$\{42.\}$

MOTIVATION BACKGROUND WHAT'S NEW CASE STUDY

Dependent Separation Protocols

$prot = \,!\, l\, x\langle l\rangle\, \{l \mapsto x\}.\, prot'$

$\{True\}$

```
let (c, c') = new_chan () in
```

$\{c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\}$

```
fork {
```

$\{c' \rightarrowtail \overline{prot}\} \vDash$

$\{c' \rightarrowtail ?\, l\, x\langle l\rangle \{l \mapsto x\}.\, \overline{prot'}\}$

```
    let l = recv c' in

    l <- !l + 2;


    send c' ()

}
```

$\{c \rightarrowtail prot\}$

```
let l = ref 40 in
```

$\{c \rightarrowtail prot * l \mapsto x * x = 40\} \vDash$

$\{c \rightarrowtail \,!\, l\, x\langle l\rangle \{l \mapsto x\}.\, prot' * l \mapsto x * x = 40\}$

```
send c l;
```

$\{c \rightarrowtail prot' * x = 40\}$

```
recv c;
```

$$\{c \rightarrowtail !\, \vec{x}:\vec{\tau}\, \langle v\rangle \{P\}.\, prot * P[\vec{t}/\vec{x}]\}\ \textsf{send}\ c\ (v[\vec{t}/\vec{x}])\ \{c \rightarrowtail prot[\vec{t}/\vec{x}]\} \qquad \text{(HT-SEND)}$$

$\{42.\}$

Dependent Separation Protocols

$prot = \,!\, l\, x \langle l \rangle \{l \mapsto x\}.\, prot'$

$\{True\}$

```
let (c, c') = new_chan () in
```

$\{c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\}$

```
fork {
```

$\{c' \rightarrowtail \overline{prot}\} \vDash$

$\{c' \rightarrowtail ?\, l\, x \langle l \rangle \{l \mapsto x\}.\, \overline{prot'}\}$

```
    let l = recv c' in
```

$\{c' \rightarrowtail \overline{prot'} * l \mapsto x\}$

```
    l <- !l + 2;


    send c' ()

}
```

$\{c \rightarrowtail prot\}$

```
let l = ref 40 in
```

$\{c \rightarrowtail prot * l \mapsto x * x = 40\} \vDash$

$\{c \rightarrowtail \,!\, l\, x \langle l \rangle \{l \mapsto x\}.\, prot' * l \mapsto x * x = 40\}$

```
send c l;
```

$\{c \rightarrowtail prot' * x = 40\}$

```
recv c;
```

$$\{c \rightarrowtail ?\, \vec{x} : \vec{\tau}\, \langle v \rangle \{P\}.\, prot\}\ \texttt{recv}\ c\ \{w.\, \exists \vec{y}.\, (w = v[\vec{y}/\vec{x}]) * c \rightarrowtail prot[\vec{y}/\vec{x}] * P[\vec{y}/\vec{x}]\}\ \text{(HT-RECV)}$$

$\{42.\}$

Dependent Separation Protocols

$prot = !\, l\, x\langle l \rangle\, \{l \mapsto x\}.\, prot'$

$\{True\}$

```
let (c, c') = new_chan () in
```

$\{c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\}$

```
fork {
```

$\{c' \rightarrowtail \overline{prot}\} \vDash$

$\{c' \rightarrowtail ?\, l\, x\langle l \rangle \{l \mapsto x\}.\, \overline{prot'}\}$

```
    let l = recv c' in
```

$\{c' \rightarrowtail \overline{prot'} * l \mapsto x\}$

```
    l <- !l + 2;
```

$\{c' \rightarrowtail \overline{prot'} * l \mapsto x + 2\}$

```
    send c' ()
}
```

$\{c \rightarrowtail prot\}$

```
let l = ref 40 in
```

$\{c \rightarrowtail prot * l \mapsto x * x = 40\} \vDash$

$\{c \rightarrowtail !\, l\, x\langle l \rangle \{l \mapsto x\}.\, prot' * l \mapsto x * x = 40\}$

```
send c l;
```

$\{c \rightarrowtail prot' * x = 40\}$

```
recv c;

!l
```

$\{42.\}$

Dependent Separation Protocols

$prot = !\, l\, x \langle l \rangle\, \{ l \mapsto x \}.\, prot'$
$prot' = ?\, l\, x \langle () \rangle\, \{ l \mapsto x + 2 \}.\, end$

$$\{True\}$$

```
let (c, c') = new_chan () in
```

$$\{c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\}$$

```
fork {
```

$$\{c' \rightarrowtail \overline{prot}\} \vDash$$

$$\{c' \rightarrowtail ?\, l\, x\langle l \rangle\{l \mapsto x\}.\, \overline{prot'}\}$$

```
    let l = recv c' in
```

$$\{c' \rightarrowtail \overline{prot'} * l \mapsto x\}$$

```
    l <- !l + 2;
```

$$\{c' \rightarrowtail \overline{prot'} * l \mapsto x + 2\} \vDash$$

$$\{c' \rightarrowtail !\, l\, x\langle () \rangle\{l \mapsto x + 2\}.\, end * l \mapsto x + 2\}$$

```
    send c' ()

}
```

$$\{c \rightarrowtail prot\}$$

```
let l = ref 40 in
```

$$\{c \rightarrowtail prot * l \mapsto x * x = 40\} \vDash$$

$$\{c \rightarrowtail !\, l\, x\langle l \rangle\{l \mapsto x\}.\, prot' * l \mapsto x * x = 40\}$$

```
send c l;
```

$$\{c \rightarrowtail prot' * x = 40\} \vDash$$

$$\{c \rightarrowtail !\, l\, x\langle () \rangle\{l \mapsto x + 2\}.\, end * x = 40\}$$

```
recv c;

!l
```

$$\{42.\}$$

Dependent Separation Protocols

$prot = !\, l\, x\langle l\rangle\, \{l \mapsto x\}.\, prot'$
$prot' = ?\, l\, x\langle()\rangle\, \{l \mapsto x + 2\}.\, end$

$\{True\}$
```
let (c, c') = new_chan () in
```
$\{c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\}$
```
fork {
```
$\{c' \rightarrowtail \overline{prot}\} \vDash$
$\{c' \rightarrowtail ?\, l\, x\langle l\rangle\{l \mapsto x\}.\, \overline{prot'}\}$
```
    let l = recv c' in
```
$\{c' \rightarrowtail \overline{prot'} * l \mapsto x\}$
```
    l <- !l + 2;
```
$\{c' \rightarrowtail \overline{prot'} * l \mapsto x + 2\} \vDash$
$\{c' \rightarrowtail !\, l\, x\langle()\rangle\{l \mapsto x + 2\}.\, end * l \mapsto x + 2\}$
```
    send c' ()
```
$\{c' \rightarrowtail end\}$
```
}
```
$\{c \rightarrowtail prot\}$
```
let l = ref 40 in
```
$\{c \rightarrowtail prot * l \mapsto x * x = 40\} \vDash$
$\{c \rightarrowtail !\, l\, x\langle l\rangle\{l \mapsto x\}.\, prot' * l \mapsto x * x = 40\}$
```
send c l;
```
$\{c \rightarrowtail prot' * x = 40\} \vDash$
$\{c \rightarrowtail !\, l\, x\langle()\rangle\{l \mapsto x + 2\}.\, end * x = 40\}$
```
recv c;

!l
```
$\{42.\}$

Dependent Separation Protocols

$prot = \,!\, l\, x\langle l\rangle\, \{l \mapsto x\}.\, prot'$
$prot' = \,?\, l\, x\langle ()\rangle\, \{l \mapsto x + 2\}.\, end$

$\{True\}$
```
let (c, c') = new_chan () in
```
$\{c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\}$
```
fork {
```
$\{c' \rightarrowtail \overline{prot}\} \vDash$
$\{c' \rightarrowtail ?\, l\, x\langle l\rangle\{l \mapsto x\}.\, \overline{prot'}\}$
```
    let l = recv c' in
```
$\{c' \rightarrowtail \overline{prot'} * l \mapsto x\}$
```
    l <- !l + 2;
```
$\{c' \rightarrowtail \overline{prot'} * l \mapsto x + 2\} \vDash$
$\{c' \rightarrowtail !\, l\, x\langle ()\rangle\{l \mapsto x + 2\}.\, end * l \mapsto x + 2\}$
```
    send c' ()
```
$\{c' \rightarrowtail end\}$
```
}
```
$\{c \rightarrowtail prot\}$
```
let l = ref 40 in
```
$\{c \rightarrowtail prot * l \mapsto x * x = 40\} \vDash$
$\{c \rightarrowtail !\, l\, x\langle l\rangle\{l \mapsto x\}.\, prot' * l \mapsto x * x = 40\}$
```
send c l;
```
$\{c \rightarrowtail prot' * x = 40\} \vDash$
$\{c \rightarrowtail !\, l\, x\langle ()\rangle\{l \mapsto x + 2\}.\, end * x = 40\}$
```
recv c;
```
$\{c \rightarrowtail end * l \mapsto x + 2 * x = 40\}$
```
!l
```

$\{42.\}$

MOTIVATION BACKGROUND WHAT'S NEW CASE STUDY

Dependent Separation Protocols

$prot = !\, l\, x\langle l\rangle\, \{l \mapsto x\}.\, prot'$
$prot' = ?\, l\, x\langle ()\rangle\, \{l \mapsto x + 2\}.\, end$

$\{True\}$

```
let (c, c') = new_chan () in
```

$\{c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\}$

```
fork {
```

$\{c' \rightarrowtail \overline{prot}\} \models$

$\{c' \rightarrowtail ?\, l\, x\langle l\rangle\{l \mapsto x\}.\, \overline{prot'}\}$

```
    let l = recv c' in
```

$\{c' \rightarrowtail \overline{prot'} * l \mapsto x\}$

```
    l <- !l + 2;
```

$\{c' \rightarrowtail \overline{prot'} * l \mapsto x + 2\} \models$

$\{c' \rightarrowtail !\, l\, x\langle ()\rangle\{l \mapsto x + 2\}.\, end * l \mapsto x + 2\}$

```
    send c' ()
```

$\{c' \rightarrowtail end\}$

```
}
```

$\{c \rightarrowtail prot\}$

```
let l = ref 40 in
```

$\{c \rightarrowtail prot * l \mapsto x * x = 40\} \models$

$\{c \rightarrowtail !\, l\, x\langle l\rangle\{l \mapsto x\}.\, prot' * l \mapsto x * x = 40\}$

```
send c l;
```

$\{c \rightarrowtail prot' * x = 40\} \models$

$\{c \rightarrowtail !\, l\, x\langle ()\rangle\{l \mapsto x + 2\}.\, end * x = 40\}$

```
recv c;
```

$\{c \rightarrowtail end * l \mapsto x + 2 * x = 40\}$

```
!l
```

$\{x + 2.\ c \rightarrowtail end * l \mapsto x + 2 * x = 40\} \models$

$\{42.\}$

MOTIVATION BACKGROUND WHAT'S NEW CASE STUDY

Dependent Separation Protocols

$prot = !\, l\, x\langle l\rangle\, \{l \mapsto x\}.\, prot'$
$prot' = ?\, l\, x\langle ()\rangle\, \{l \mapsto x + 2\}.\, end$

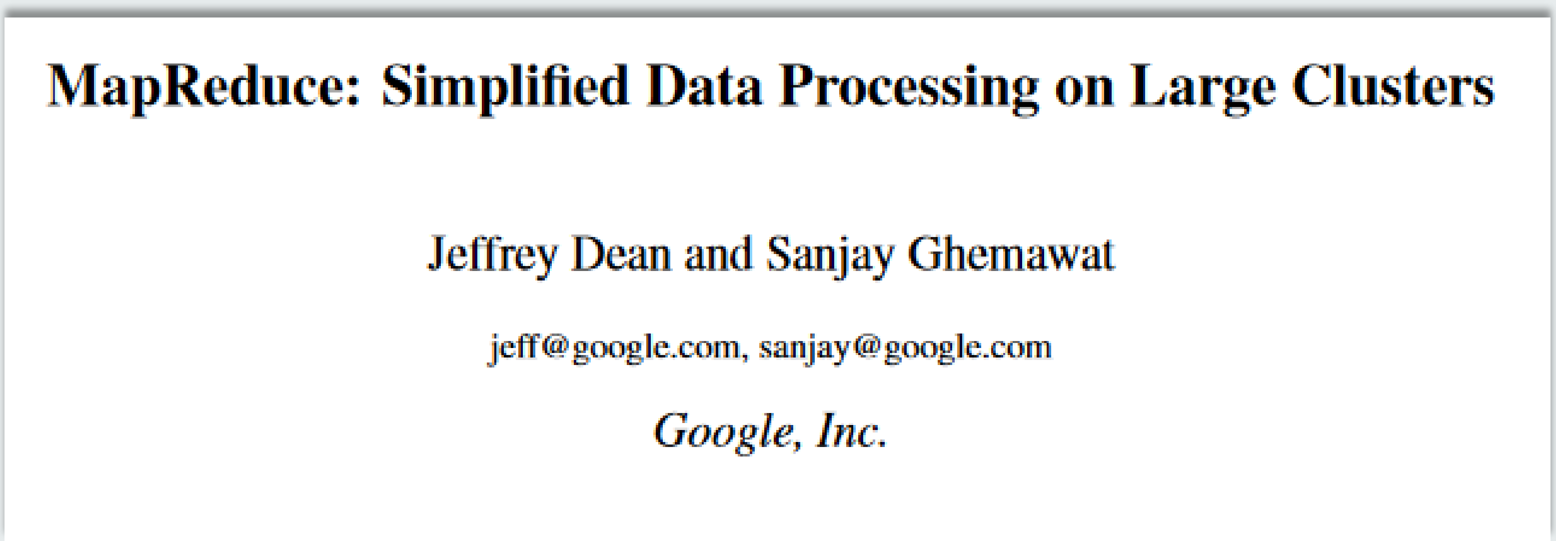
- **Safety:** The program will not get stuck
- **Postcondition validity:** If the program terminates, the postcondition holds

$\{True\}$
```
let (c, c') = new_chan () in
```
$\{c \rightarrowtail prot * c' \rightarrowtail \overline{prot}\}$
```
fork {
```
$\{c' \rightarrowtail \overline{prot}\} \vDash$
$\{c' \rightarrowtail ?\, l\, x\langle l\rangle\{l \mapsto x\}.\overline{prot'}\}$
```
    let l = recv c' in
```
$\{c' \rightarrowtail \overline{prot'} * l \mapsto x\}$
```
    l <- !l + 2;
```
$\{c' \rightarrowtail \overline{prot'} * l \mapsto x + 2\} \vDash$
$\{c' \rightarrowtail !\, l\, x\langle ()\rangle\{l \mapsto x + 2\}.\, end * l \mapsto x + 2\}$
```
    send c' ()
```
$\{c' \rightarrowtail end\}$
```
}
```
$\{c \rightarrowtail prot\}$
```
let l = ref 40 in
```
$\{c \rightarrowtail prot * l \mapsto x * x = 40\} \vDash$
$\{c \rightarrowtail !\, l\, x\langle l\rangle\{l \mapsto x\}.\, prot' * l \mapsto x * x = 40\}$
```
send c l;
```
$\{c \rightarrowtail prot' * x = 40\} \vDash$
$\{c \rightarrowtail !\, l\, x\langle ()\rangle\{l \mapsto x + 2\}.\, end * x = 40\}$
```
recv c;
```
$\{c \rightarrowtail end * l \mapsto x + 2 * x = 40\}$
```
!l
```
$\{x + 2.\ c \rightarrowtail end * l \mapsto x + 2 * x = 40\} \vDash$
$\{42.\}$

Map-Reduce

«Hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters everyday.»

Dean and Ghemawat [2004]



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Map-Reduce

- [Mephisto, Mephisto, Faust]
- [Gretchen, Faust ,Faust]

flatMap f

group

- (Mephisto, 2)
- (Faust, 1)
- (Gretchen, 1)
- (Faust, 2)

- (Mephisto, [2])
- (Faust, [1, 2])
- (Gretchen, [1])

flatMap g

end

- (Mephisto, 2)
- (Faust, 3)
- (Gretchen, 1)

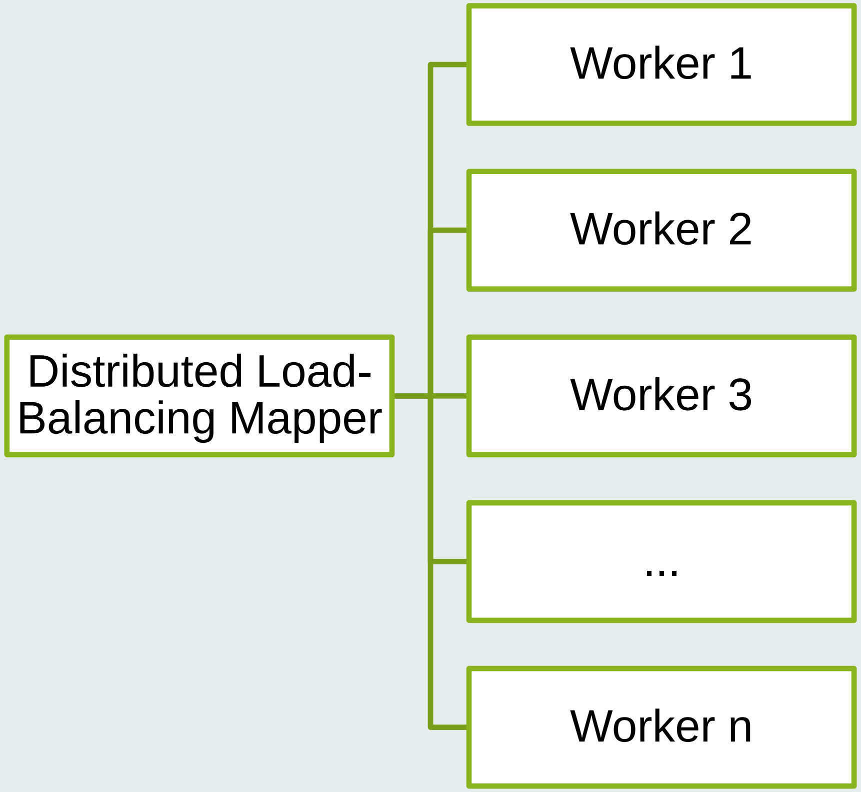
«map»

$f\colon String \to List\ (String * \mathbb{N})$

«reduce»

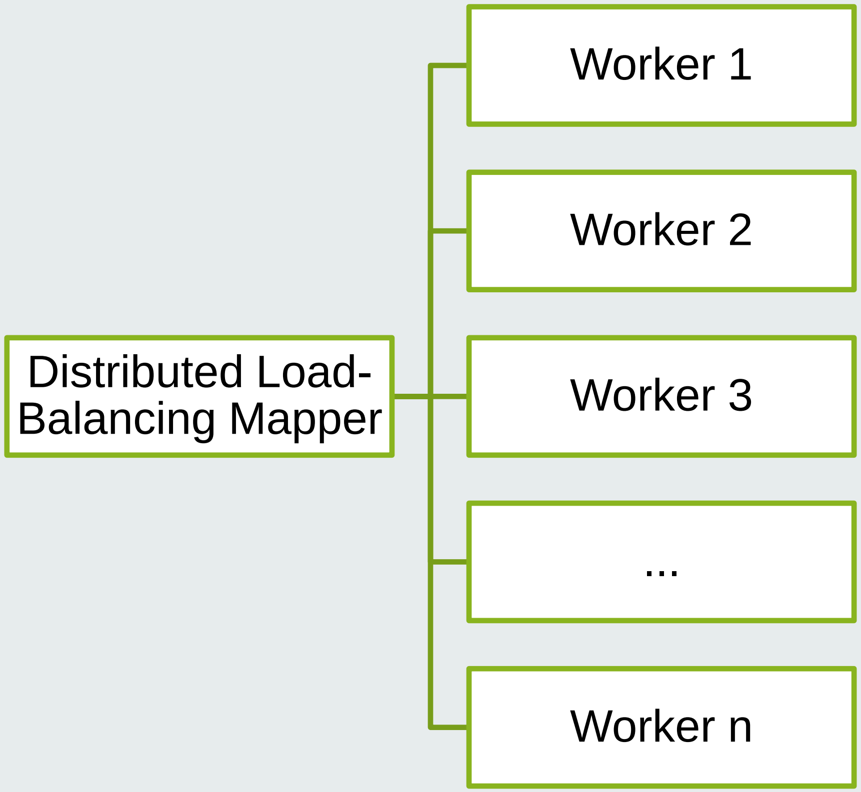
$g\colon (String * List\ \mathbb{N}) \to List\ (String * \mathbb{N})$

Map-Reduce



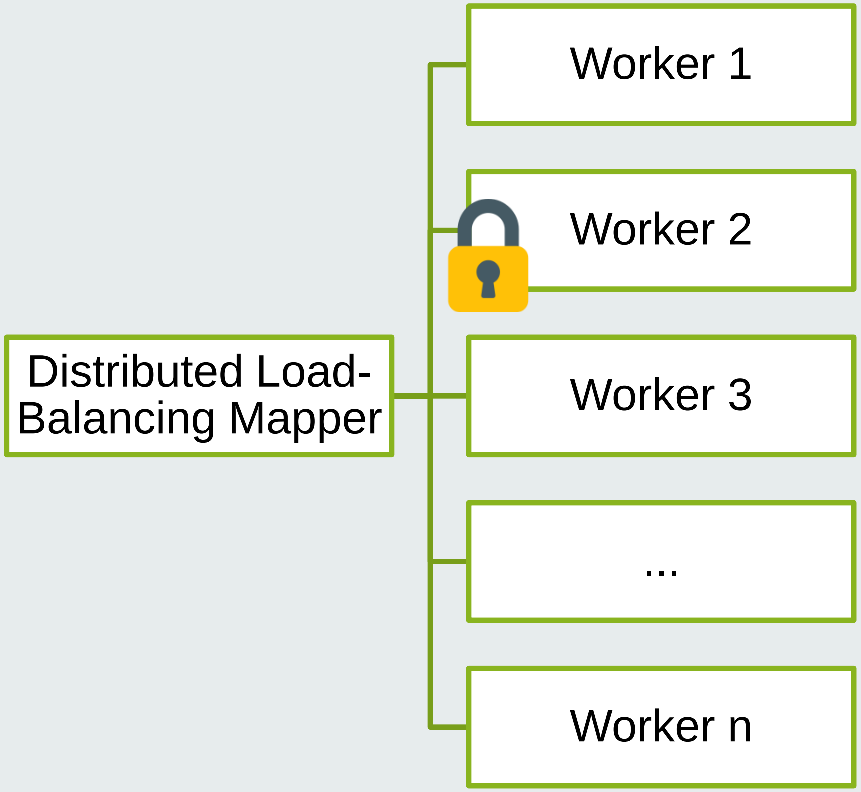
- Load-balancing mapper sends some work to be mapped

Map-Reduce



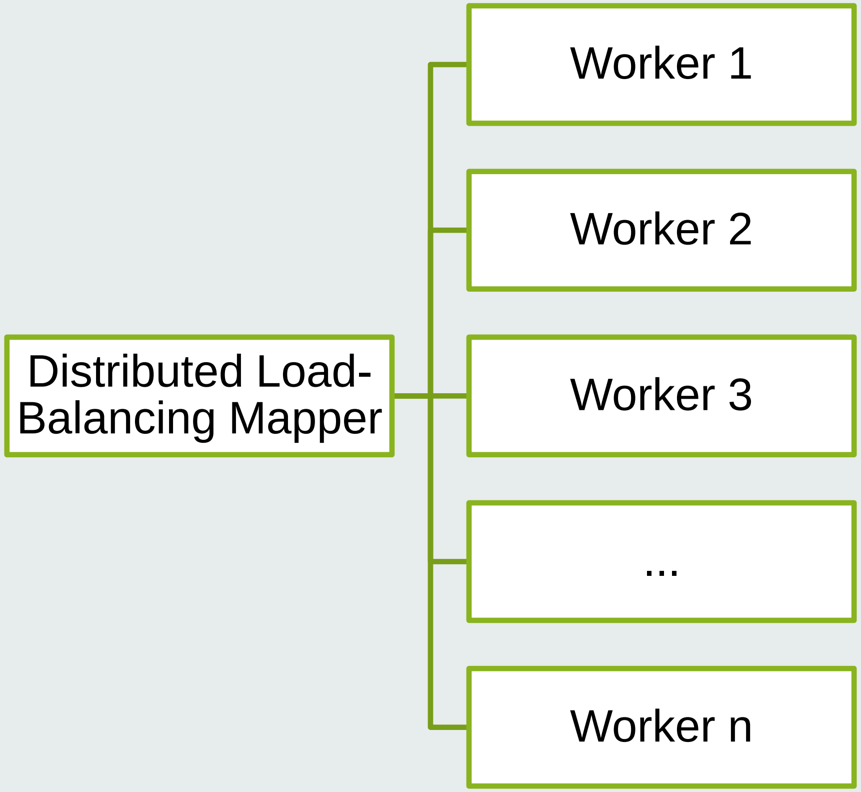
- Load-balancing mapper sends some work to be mapped
- Same channel endpoint used by all the workers («manifest sharing»)

Map-Reduce



- Load-balancing mapper sends some work to be mapped
- Same channel endpoint used by all the workers («manifest sharing»)
- Some worker acquires the lock on the channel endpoint, receives the work and releases the lock

Map-Reduce



- Load-balancing mapper sends some work to be mapped
- Same channel endpoint used by all the workers («manifest sharing»)
- Some worker acqires the lock on the channel endpoint, receives the work and releases the lock
- **Support for multiple concurrency protocols needed**

Map-Reduce

- Previous work was able to verify this too, but **only for specific mappers and reducers**
- With Actris, verification is possible for **general mappers and reducers**
- Actris is expressive enough to talk about complex mixed concurrency models, e.g. endpoint sharing using locks

	Lines of Code
Implementation	165
Proof	627

Summary

- Support for programs that use a **combination of concurrency paradigms**

Summary

- Support for programs that use a **combination of concurrency paradigms**
- Use **dependent separation protocols**
  - Uses **separation logic**
  - Inspired by **session types**

Summary

- Support for programs that use a **combination of concurrency paradigms**
- Use **dependent separation protocols**
  - Uses **separation logic**
  - Inspired by **session types**
- Prove **functional correctness of Map-Reduce**

Summary

- Support for programs that use a **combination of concurrency paradigms**
- Use **dependent separation protocols**
  - Uses **separation logic**
  - Inspired by **session types**
- Prove **functional correctness of Map-Reduce**
- Future Work
  - Multy-party dependent separation protocols
  - Proofs of deadlock-freedom