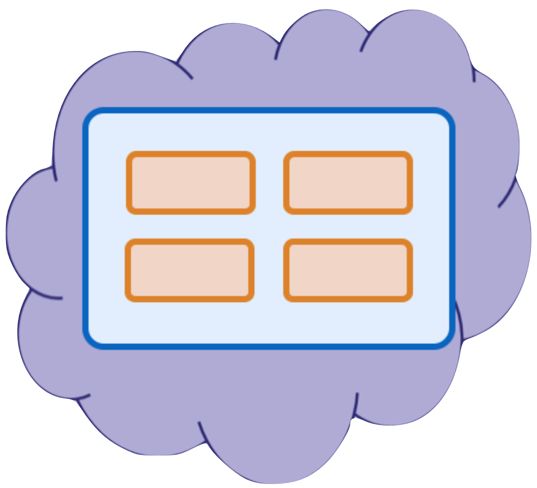
Programming and Proving with Distributed Protocols

Disel: Distributed Separation Logic



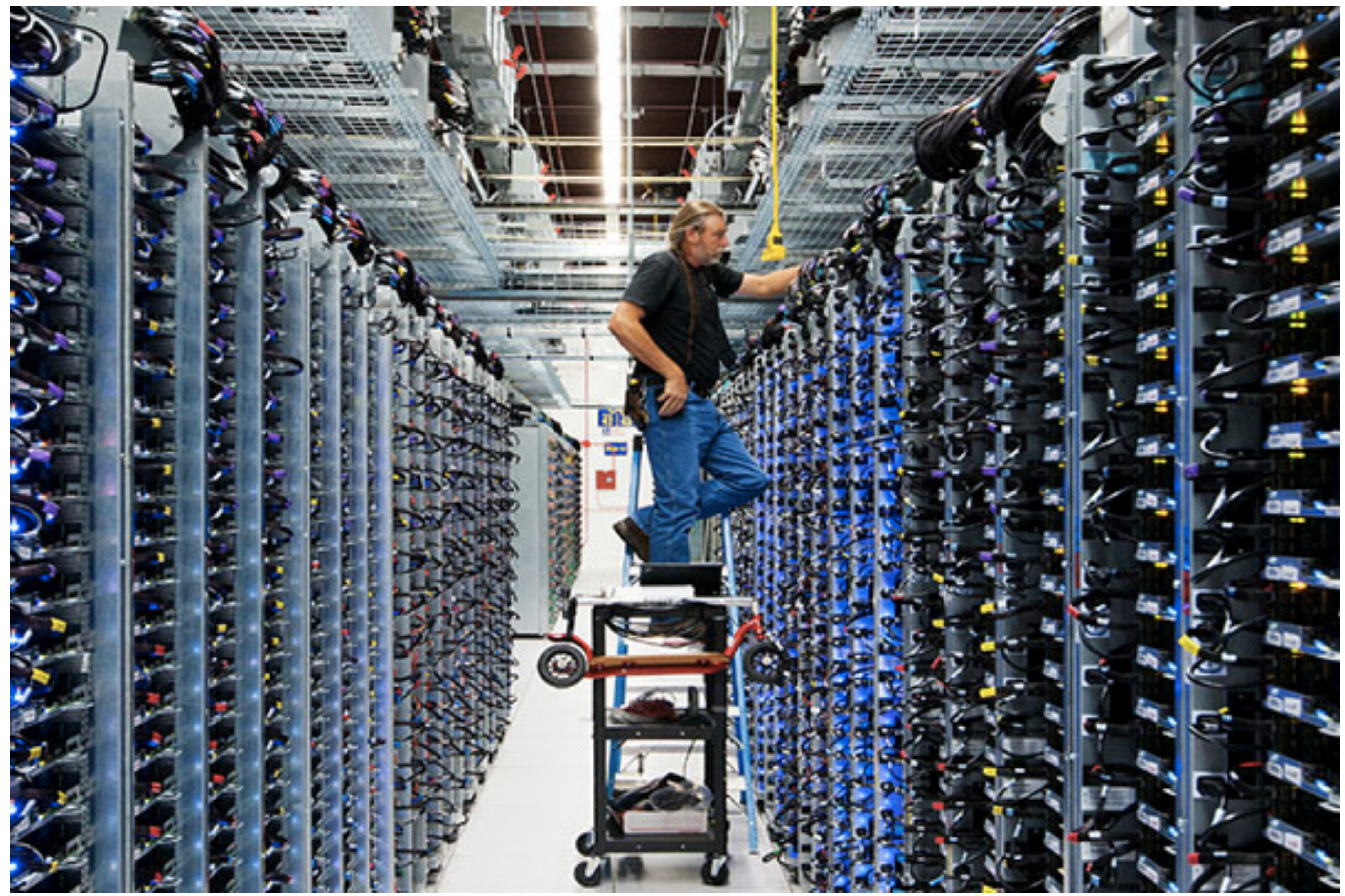
$$\vdash \{P\}\ c\ \{Q\}$$

`http://distributedcomponents.net`

Ilya Sergey **James R. Wilcox** Zach Tatlock

UCL PLSE W PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING

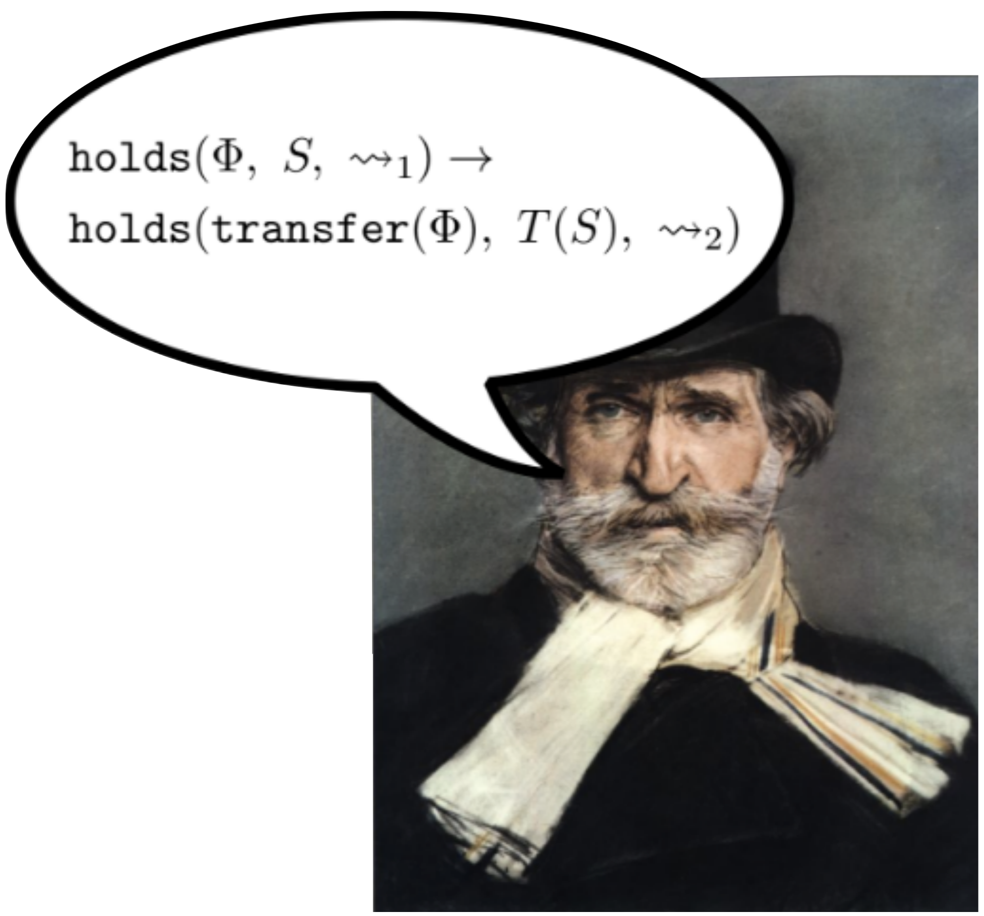
Distributed Systems

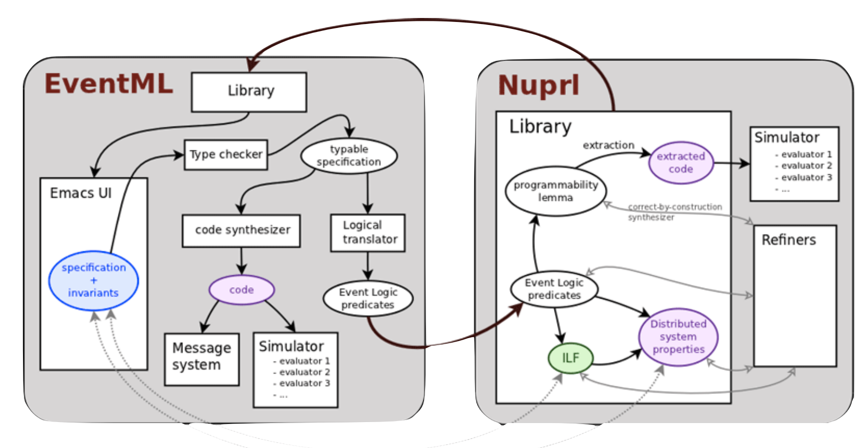


Distributed *Infrastructure*

Distributed *Applications*

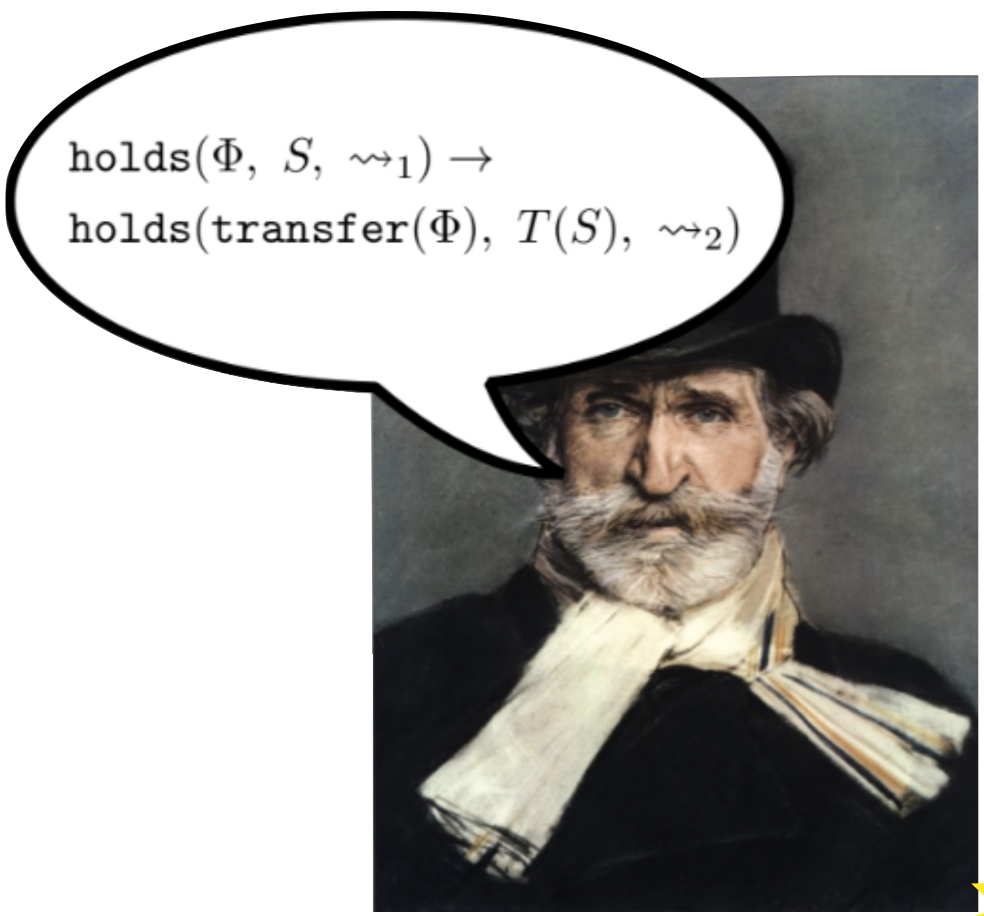
Verified Distributed Systems





Verified Distributed *Infrastructure*

Verified Distributed *Applications*



Verified Distributed *Applications*

Challenging to verify apps in terms of infra.

starting from scratch is unacceptable

Indicates deeper problems with composition

one node's client is another's server!

Challenges

Client reasoning

Invariants

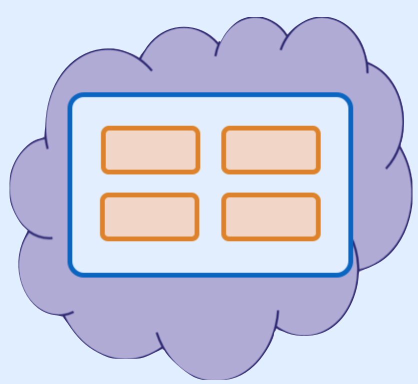
Separation

Solutions

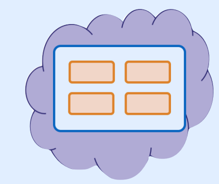
Protocols

WithInv rule

Frame rule/Hooks

Disel:  $\vdash \{P\}\ c\ \{Q\}$

Outline

 $\vdash \{P\}\ c\ \{Q\}$

Protocols and running example

Logical mechanisms

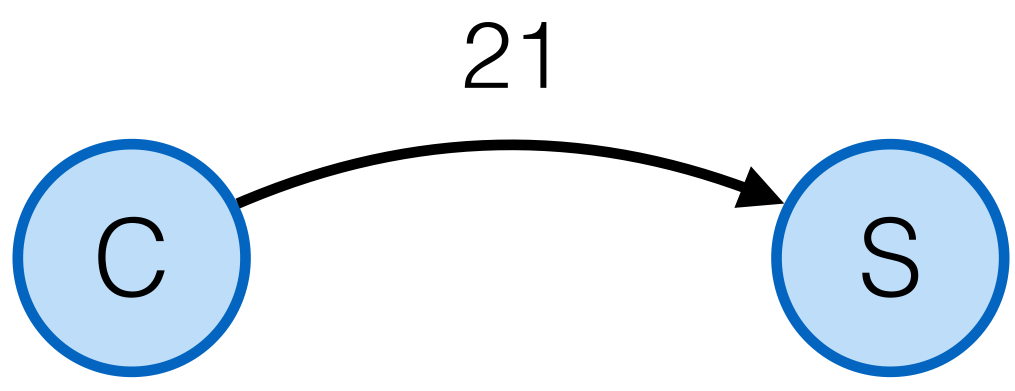
programming with protocols

invariants

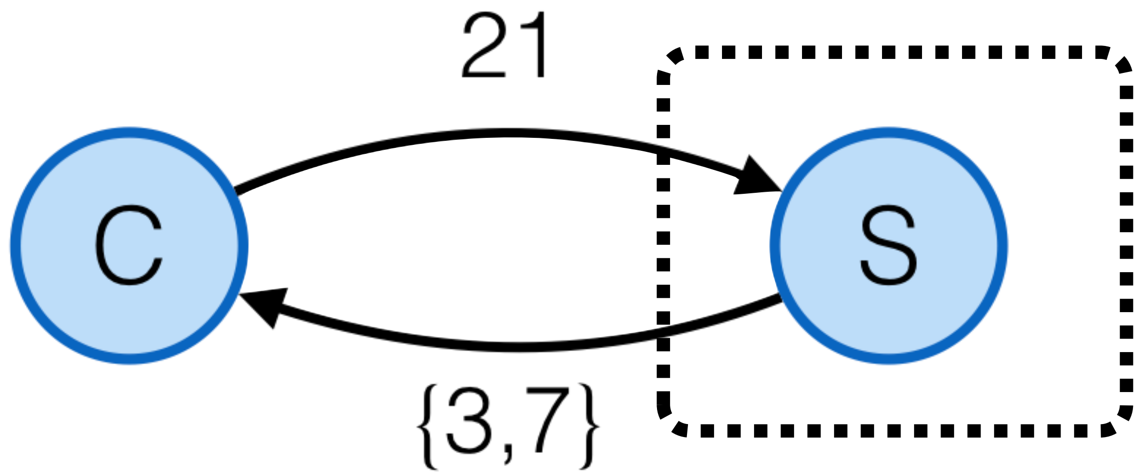
framing and hooks

Implementation and future work

Cloud Compute



Cloud Compute



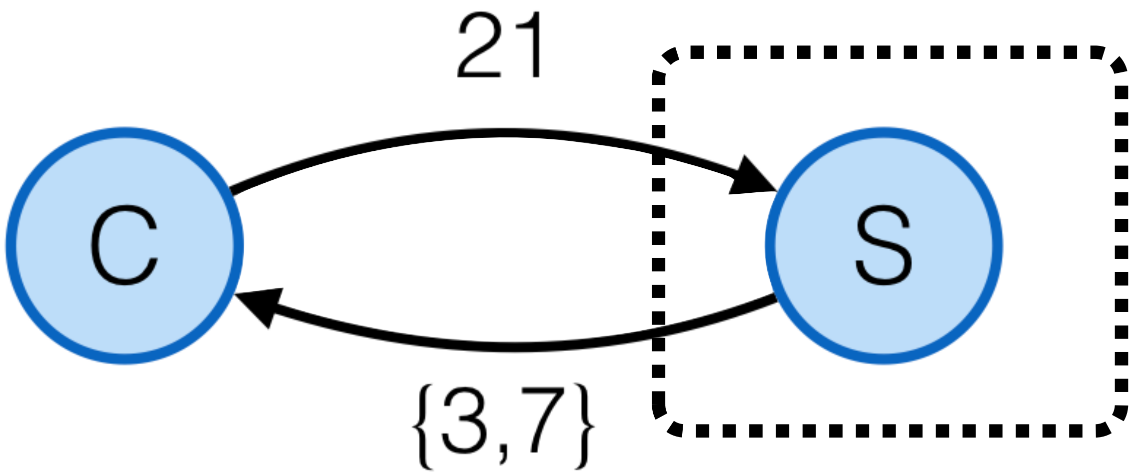
Cloud Compute: Server

```
while true:
  (from, n) <- recv Req
  send Resp(n, factors(n)) to from
```

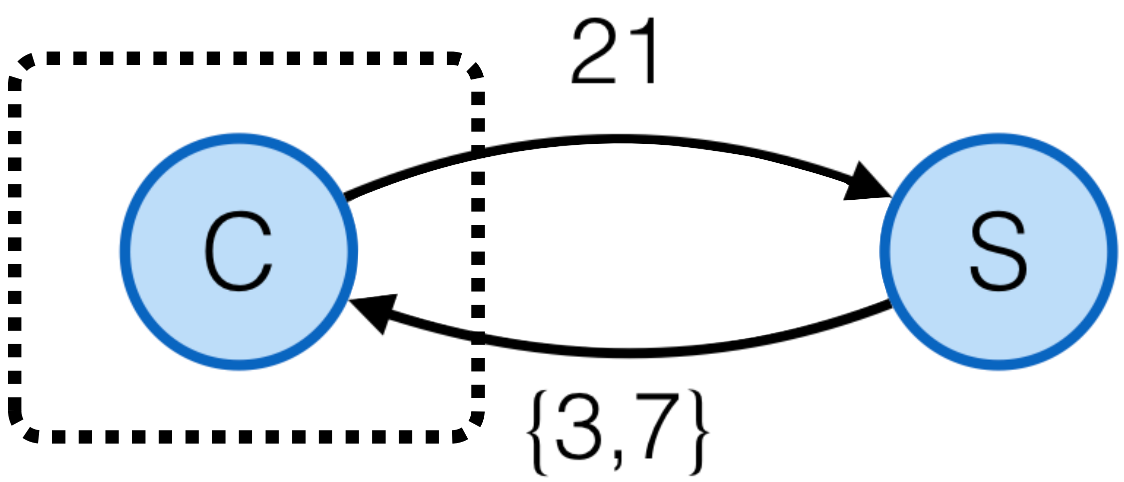
Traditional specification:
messages from server have correct factors

Proved by finding an invariant of the system

Cloud Compute: Server



Cloud Compute: Client



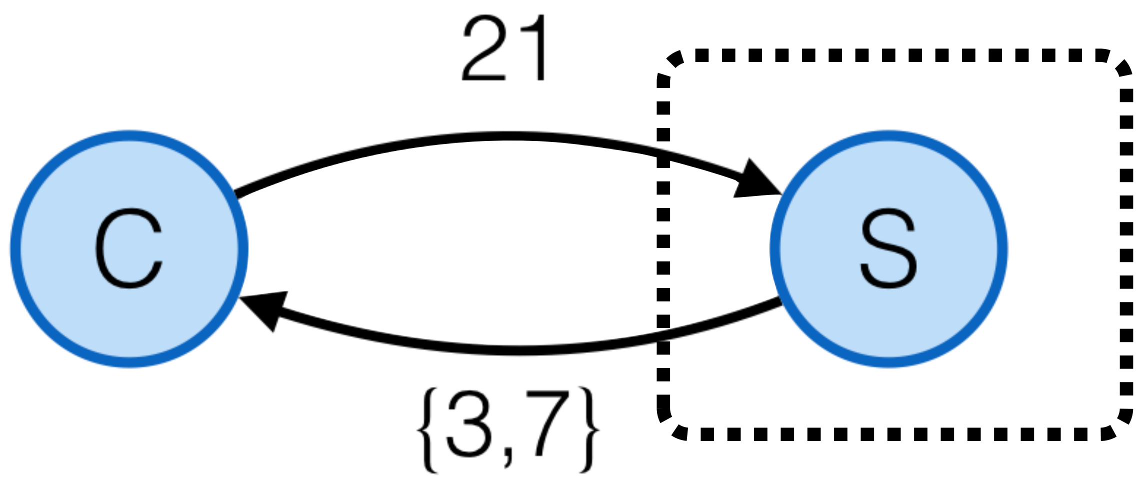
Cloud Compute: Client

```
send Req(21) to server
(_, ans) <- recv Resp
assert ans == {3, 7}
```

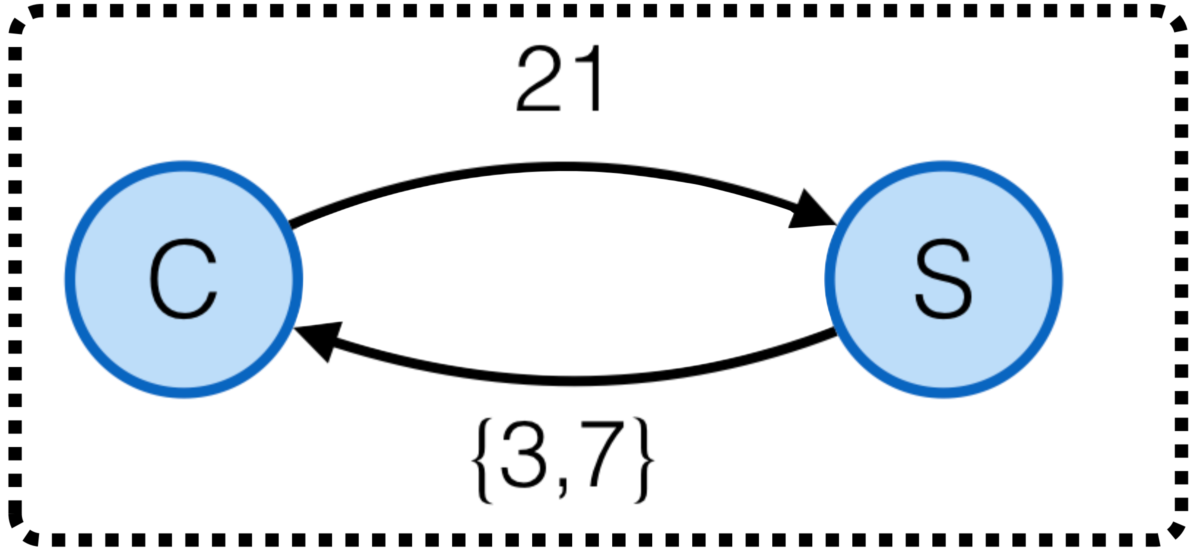
Start over with clients in system?

In Disel: use protocol to describe client interface

Protocols



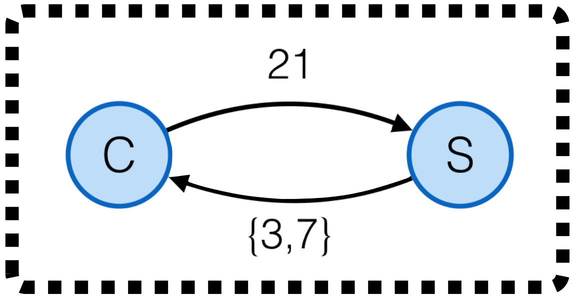
Protocols



A protocol is an **interface** among nodes

Enables compositional verification

Cloud Compute Protocol



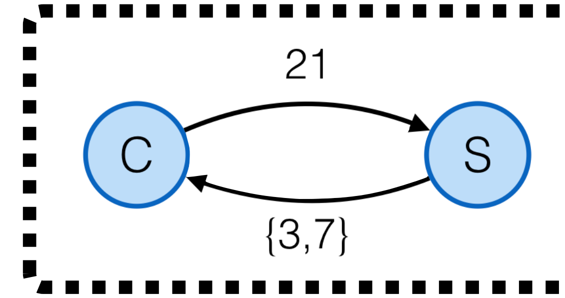
Messages:

State:

Transitions:

- Sends: precondition and effect
- Receives: effect

Cloud Compute Protocol



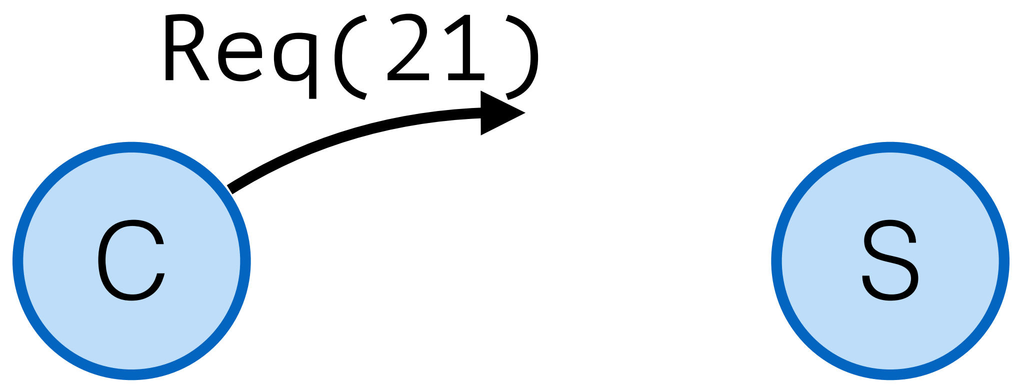
Messages: `Req(n) | Resp(n,s)`

State: `outstanding: Set<Msg>`

Transitions:

Sends:	Req	Resp
Receives:	Req	Resp

Cloud Compute

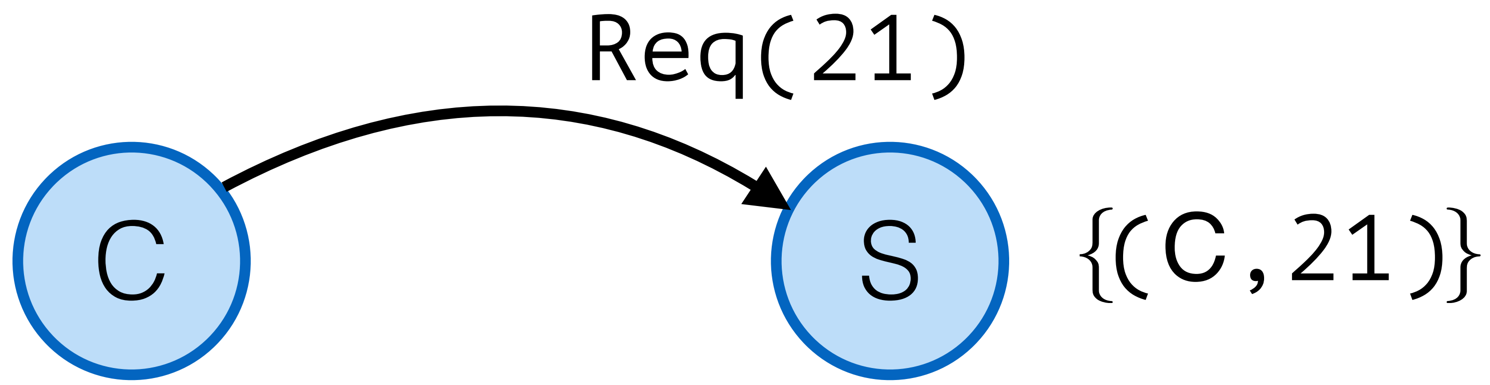


Send **`Req(n)`**

Precondition: none

Effect: none

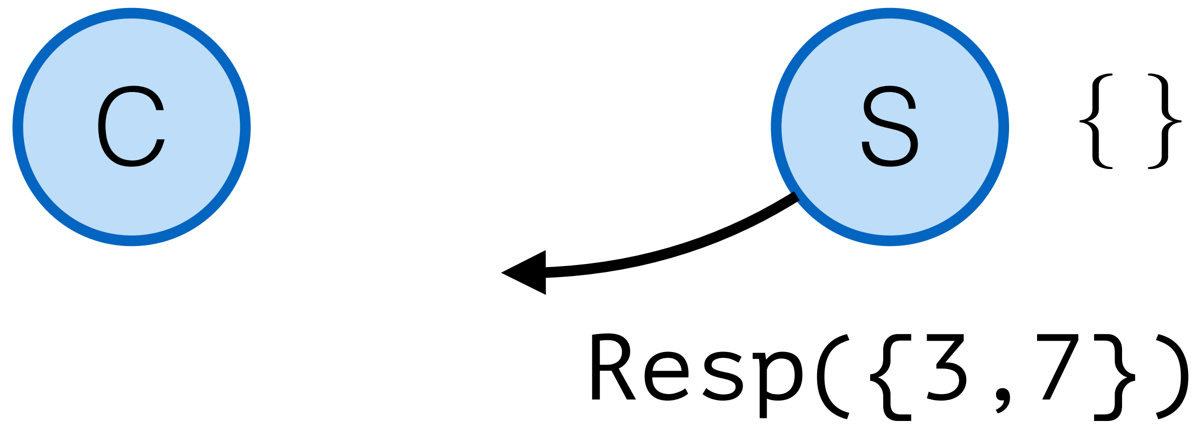
Cloud Compute



Receive `Req(n)`

Effect: add `(from, n)` to `out`

Cloud Compute



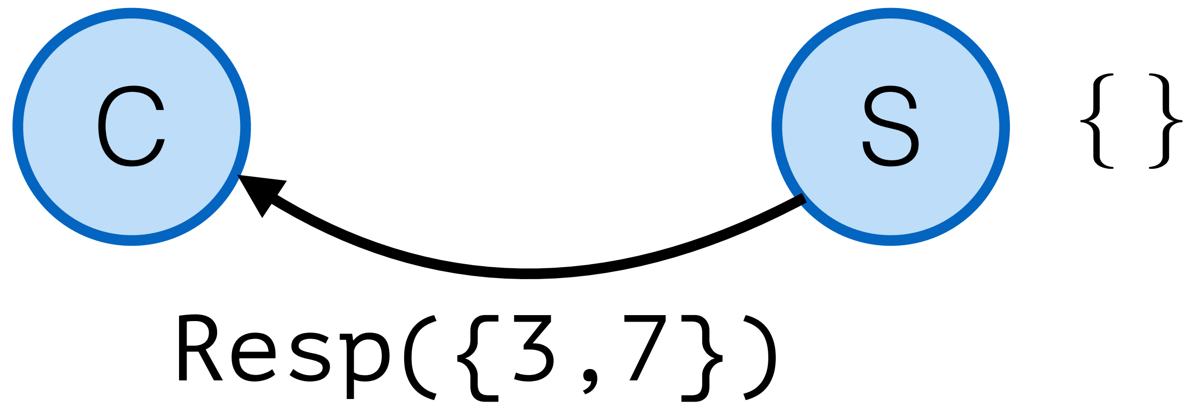
Send `Resp(n,l)`

Requires: `l == factors(n)`

`(n,to)` in `out`

Effect: removes `(n,to)` from `out`

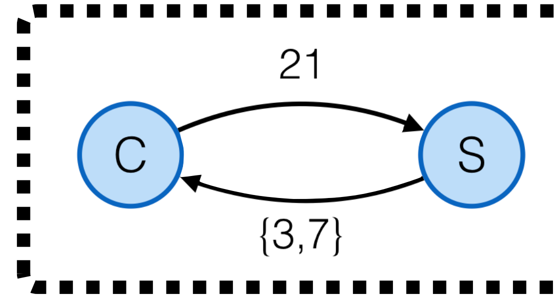
Cloud Compute



Recv `Resp(n,l)`

Effect: none

Cloud Compute Protocol



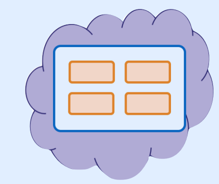
Messages: `Req(n) | Resp(n,s)`

State: `outstanding: Set<Msg>`

Transitions:

Sends:	`Req`	`Resp`
Receives:	`Req`	`Resp`

Outline

 $\vdash \{P\}\ c\ \{Q\}$

Protocols and running example

Logical mechanisms

programming with protocols

invariants

framing and hooks

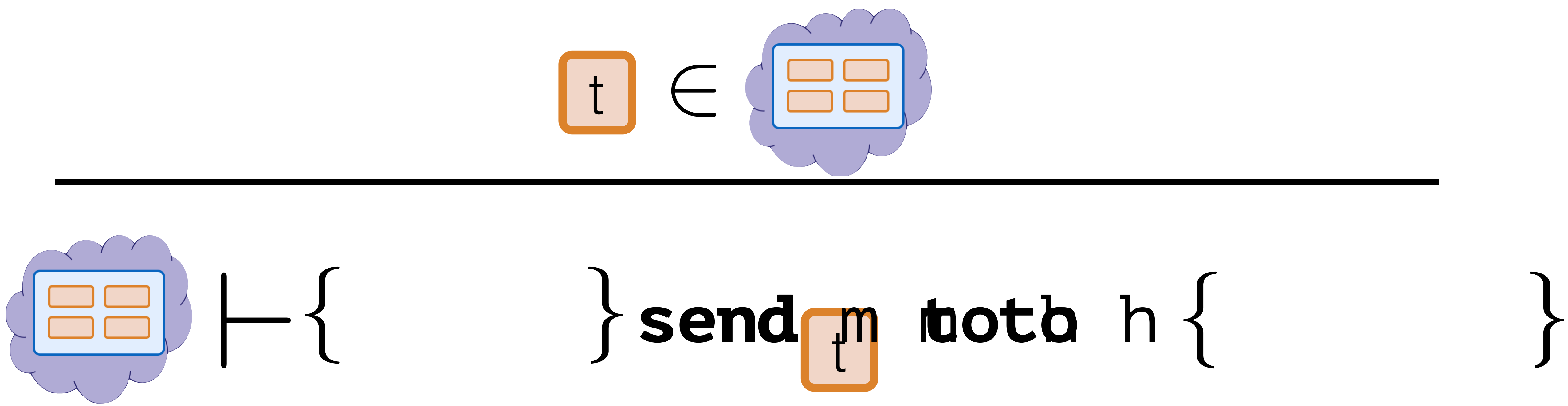
Implementation and future work

Cloud Compute: Server

```
while true:
  (from, n) <- recv Req
  send Resp(n, factors(n)) to from
```

Precondition on **send** requires correct factors

Cloud Compute: Server




```
while true:
  (from, n) <- recv Req
  send Resp(n, factors(n)) to from
```

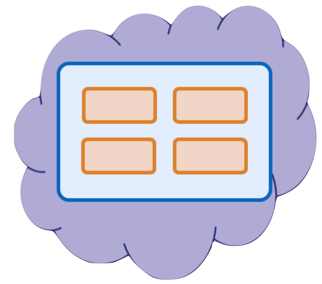
Precondition on **send** requires correct factors

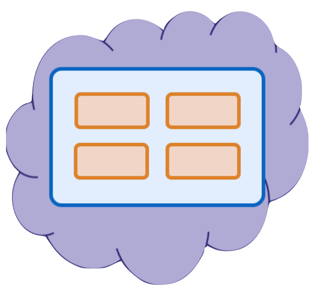
Cloud Compute: Client

```
send Req(21) to server
(_, ans) <- recv Resp
assert ans == {3, 7}
```

recv doesn't ensure correct factors

Cloud Compute: Client

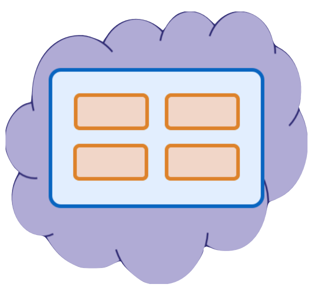
$\boxed{\text{t}} \in$ 

 $\vdash \{\top\}\ \mathbf{recv}_{\boxed{\text{t}}}\ \mathtt{m}\ \{recvd(\mathtt{m})\}$

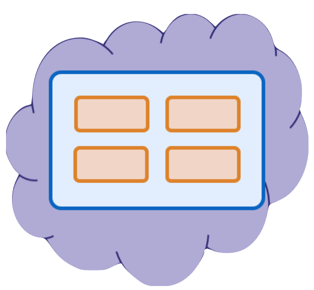
```
send Req(21) to server
(_, ans) <- recv Resp
assert ans == {3, 7}
```

recv doesn’t ensure correct factors

Protocol Invariants

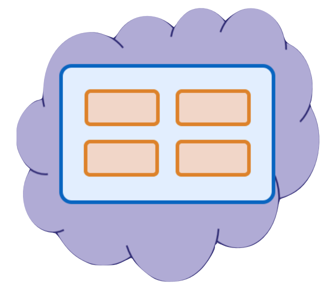
 $\vdash \{P\}\ c\ \{Q\}$ $\qquad I$ inductive

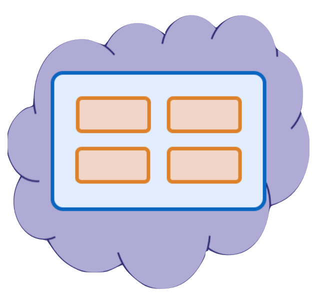
---

 $' \vdash \{P \wedge I\}\ c\ \{Q \wedge I\}$

Protocol where every state satisfies I

Cloud Compute: Client

$\boxed{\text{t}} \in$ 

 $' \vdash \{\top\}\ \mathbf{recv}_{\boxed{\text{t}}}\ \mathtt{m}\ \{\mathit{recvd}(\mathtt{m})\}$

```
send Req(21) to server
(_, ans) <- recv Resp
assert ans == {3, 7}
```

Now **recv** ensures correct factors

Cloud Compute: More Clients

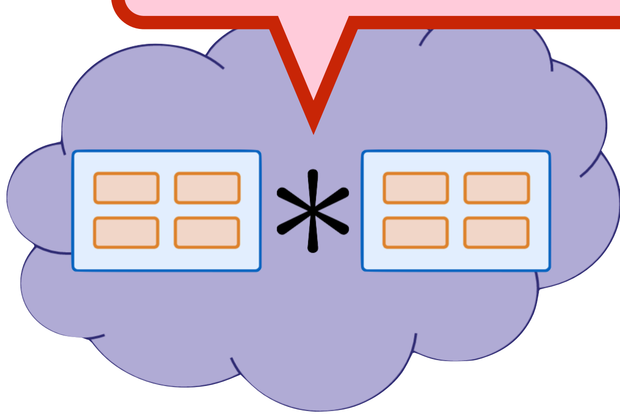
```
send Req(21) to server₁
send Req(35) to server₂
(_, ans₁) <- recv Resp
(_, ans₂) <- recv Resp
assert ans₁ ∪ ans₂ == {3, 5, 7}
```

Same protocol enables verification

Frame rule

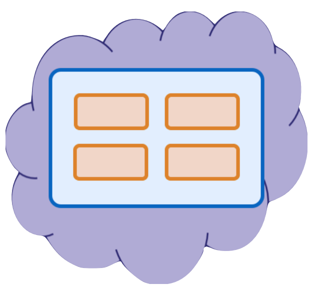
 $\vdash \{P\}\ c\ \{Q\}$ $\qquad R$ stable

independent protocols

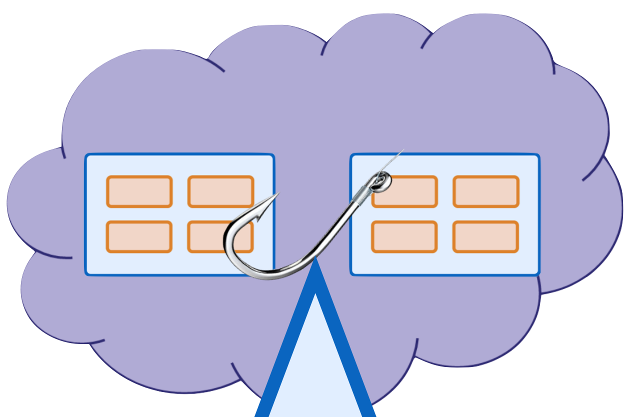
 $\vdash \{P * R\}\ c\ \{Q * R\}$

Reuse invariants from component protocols

Frame rule: Hooks

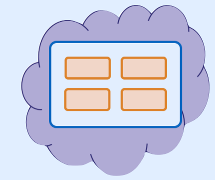
 $\vdash \{P\}\ c\ \{Q\}$ $\qquad R$ stable

---

 $\vdash \{P * R\}\ c\ \{Q * R\}$

Allows one protocol to restrict another

Outline

 $\vdash \{P\}\ c\ \{Q\}$

Protocols and running example

Logical mechanisms

programming with protocols

invariants

framing and hooks

Implementation and future work

Implementation

Shallowly embedded in Coq

with full power of functional programming

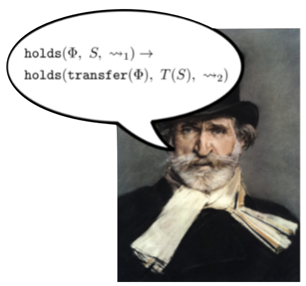
Executable via extraction to OCaml

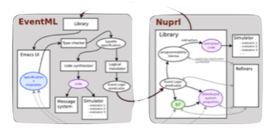
via trusted shim to implement semantics

Case study: two-phase commit

exercises all features of the logic

Related and Future Work





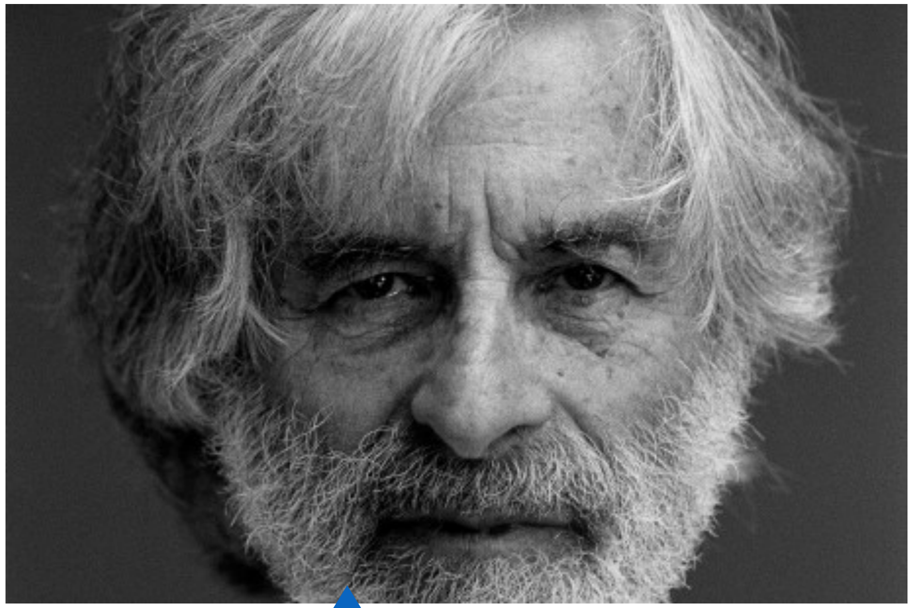
Concurrent separation logics

Iris, FCSL, CAP, …

Adding other effects

e.g. mutable heap, threads, failure…

Composition: A way to make proofs harder



"In 1997, the unfortunate reality is that engineers rarely specify and reason formally about the systems they build. It seems unlikely that reasoning about the composition of open-system specifications will be a practical concern within the next 15 years."

Challenges

Client reasoning

Invariants

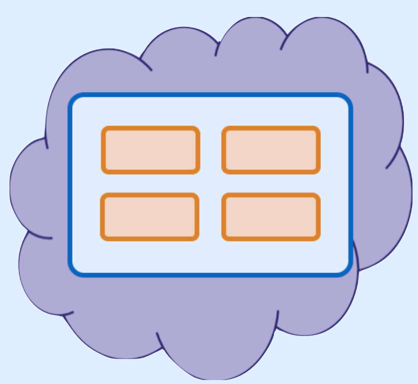
Separation

Solutions

Protocols

WithInv rule

Frame rule/Hooks

Disel:  $\vdash \{P\}\ c\ \{Q\}$