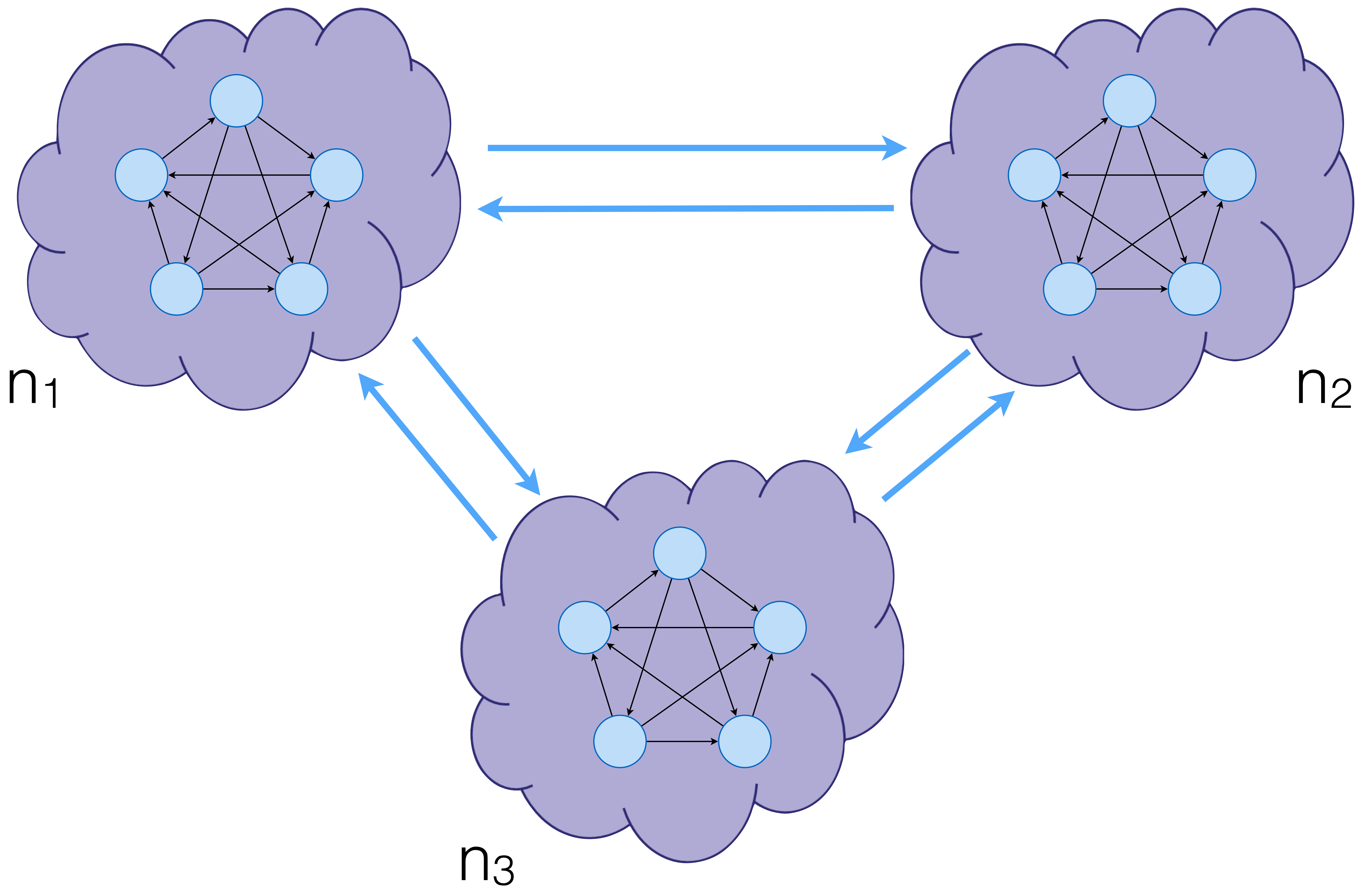
Programming and Proving with Distributed Protocols

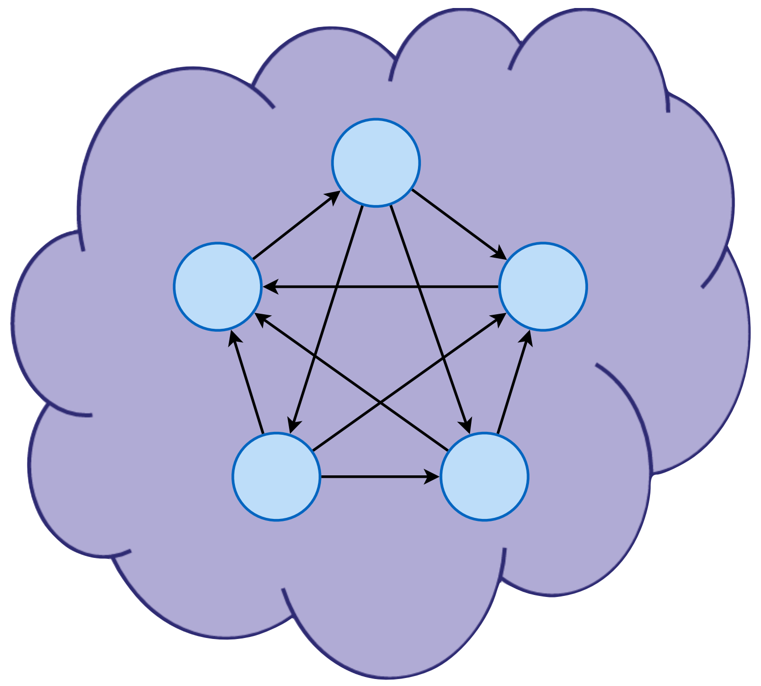
Ilya Sergey

UCL

James R. Wilcox

Zachary Tatlock

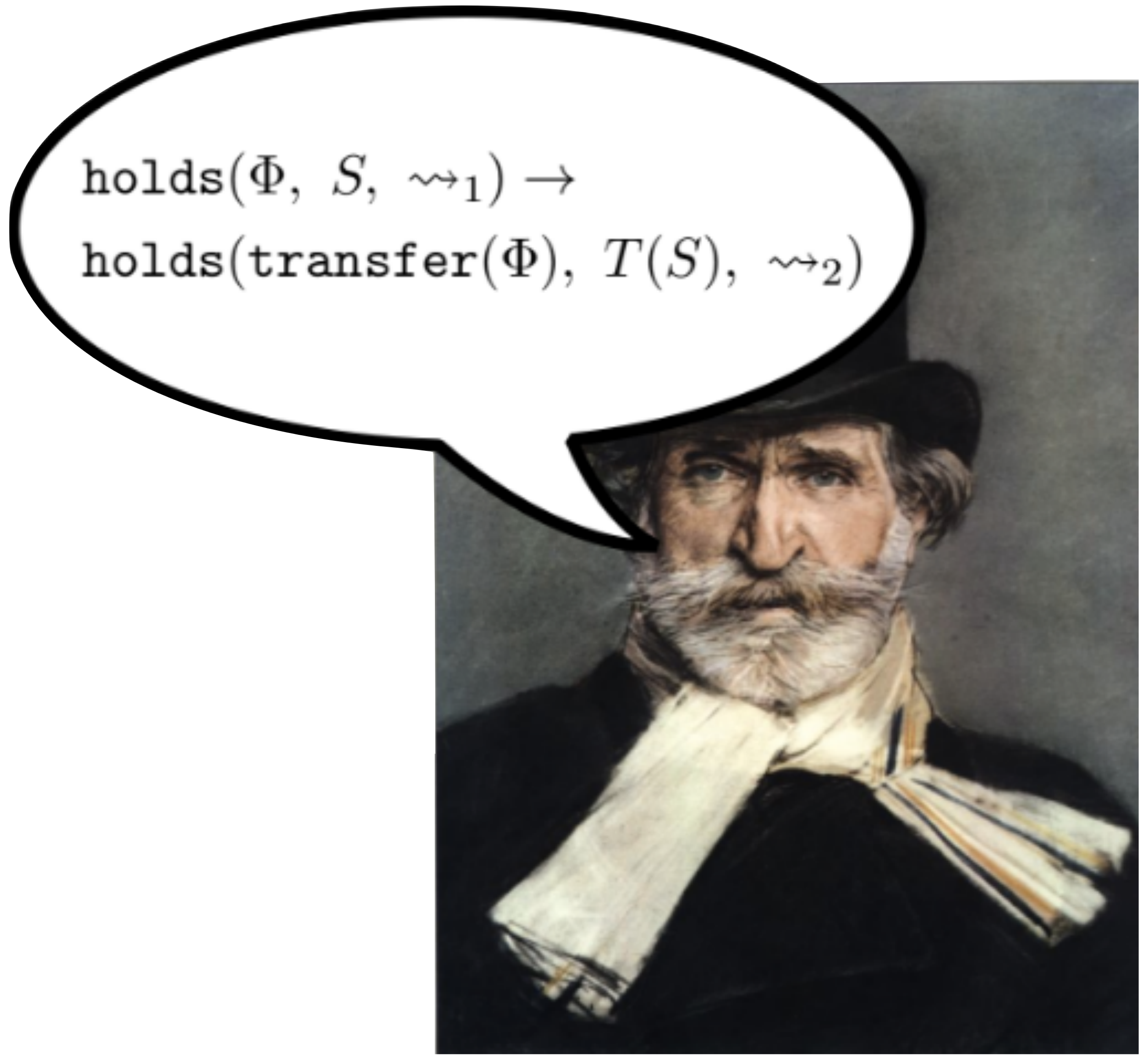

n_1
n_2
n_3

protocol P = 

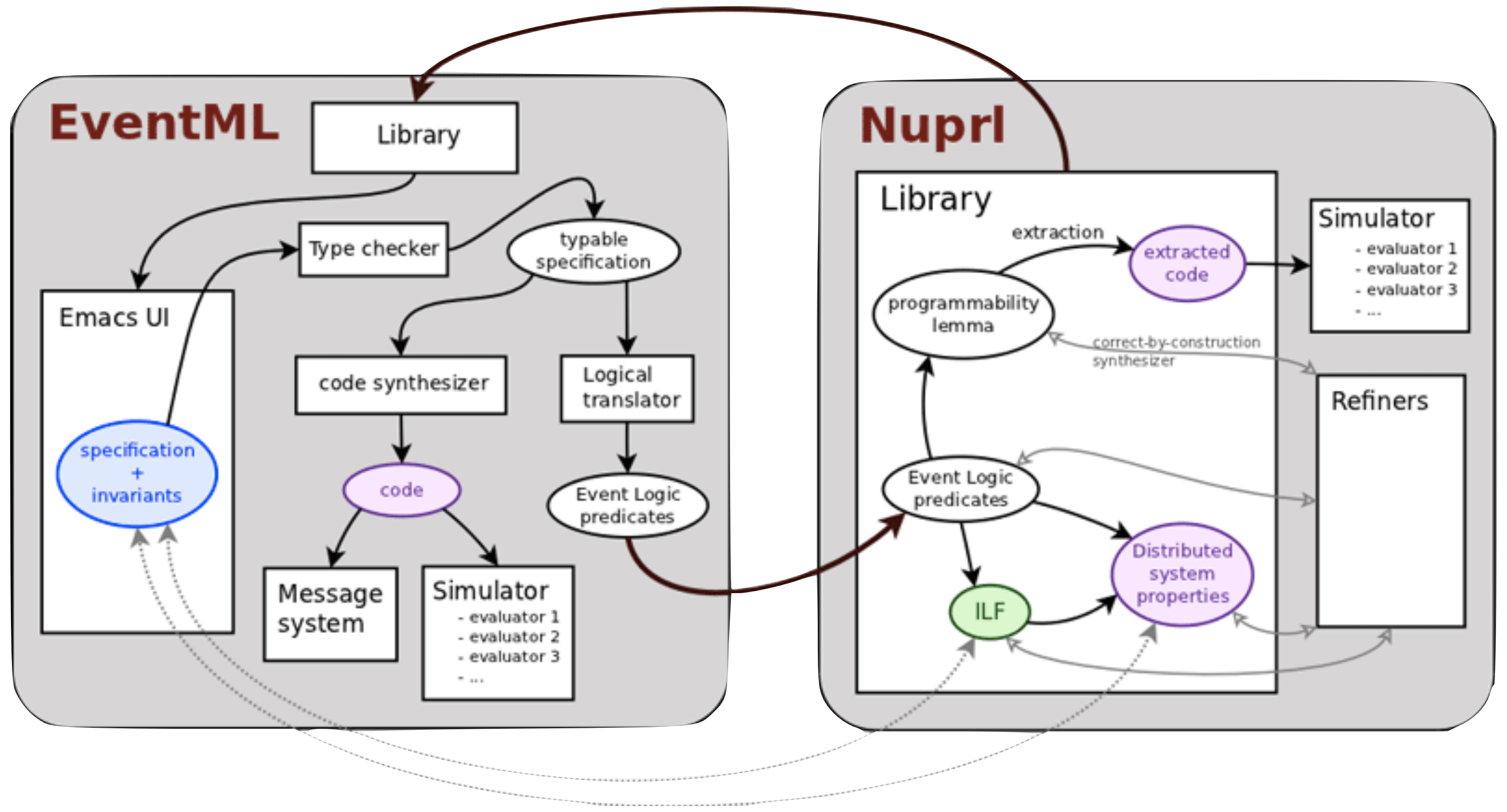
- **Invariants**: $Inv(s_0) \wedge \ \forall\, s\ s',\ Inv(s) \wedge transfer\,(P, s, s') \Rightarrow Inv(s')$
- **Liveness**: $\square(Q(\text{state of } n_1) \ \Rightarrow \diamond(R(\text{state of } n_2)))$
- **Refinement**: P ("implementation") ≤ P′ ("specification")

Verified Distributed Systems

- Invariants ✓
- Liveness ✓
- Refinement ✓



PSync



- Asynchronous updates
- Message reordering
- Packet loss
- Node crashes
- Network partitions
- Reconfiguration
- Byzantine faults

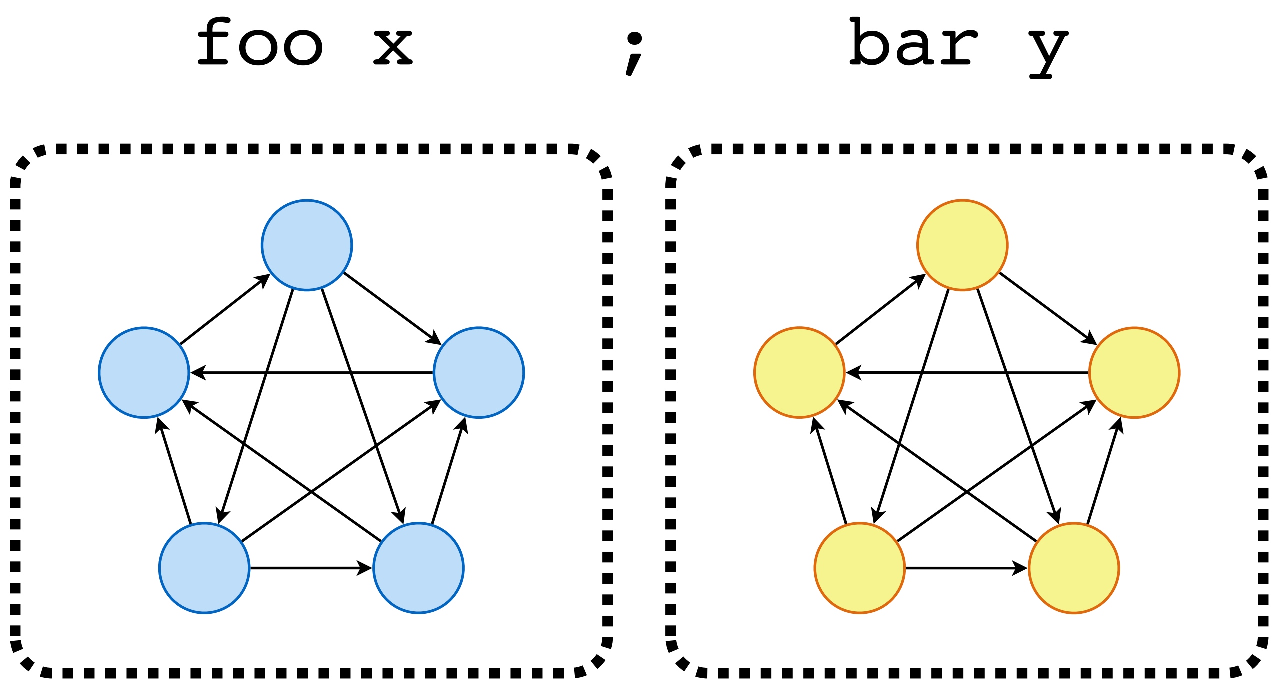
- Invariants
- Liveness
- Refinement
- Composition
  (*aka* Reusability)

Composition in Distributed Systems

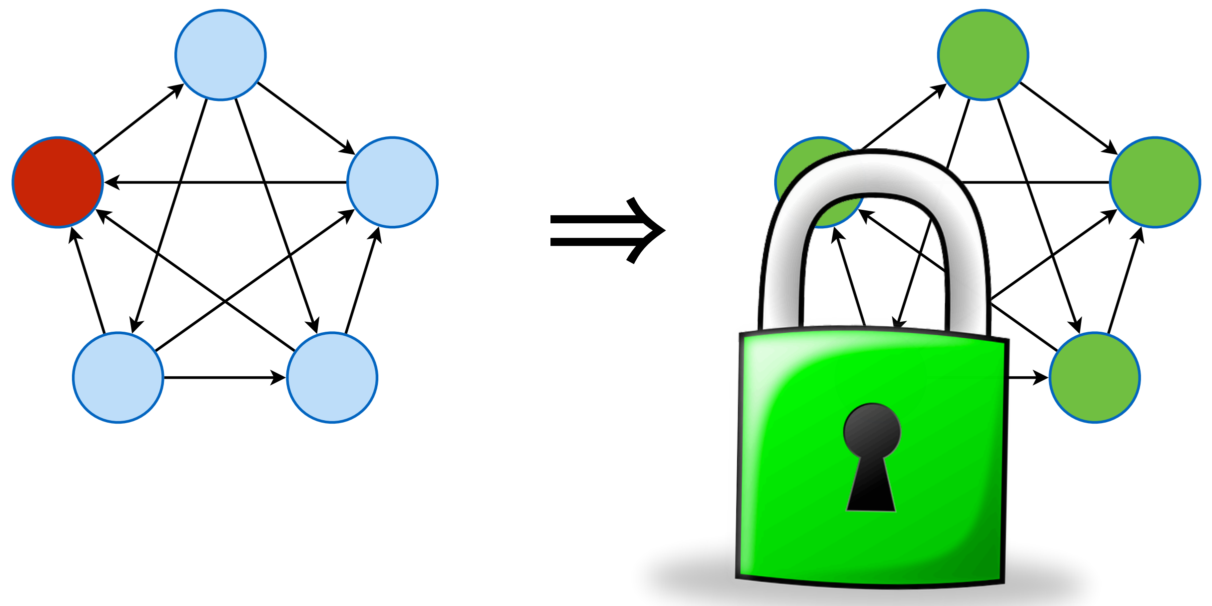
- Modular program verification

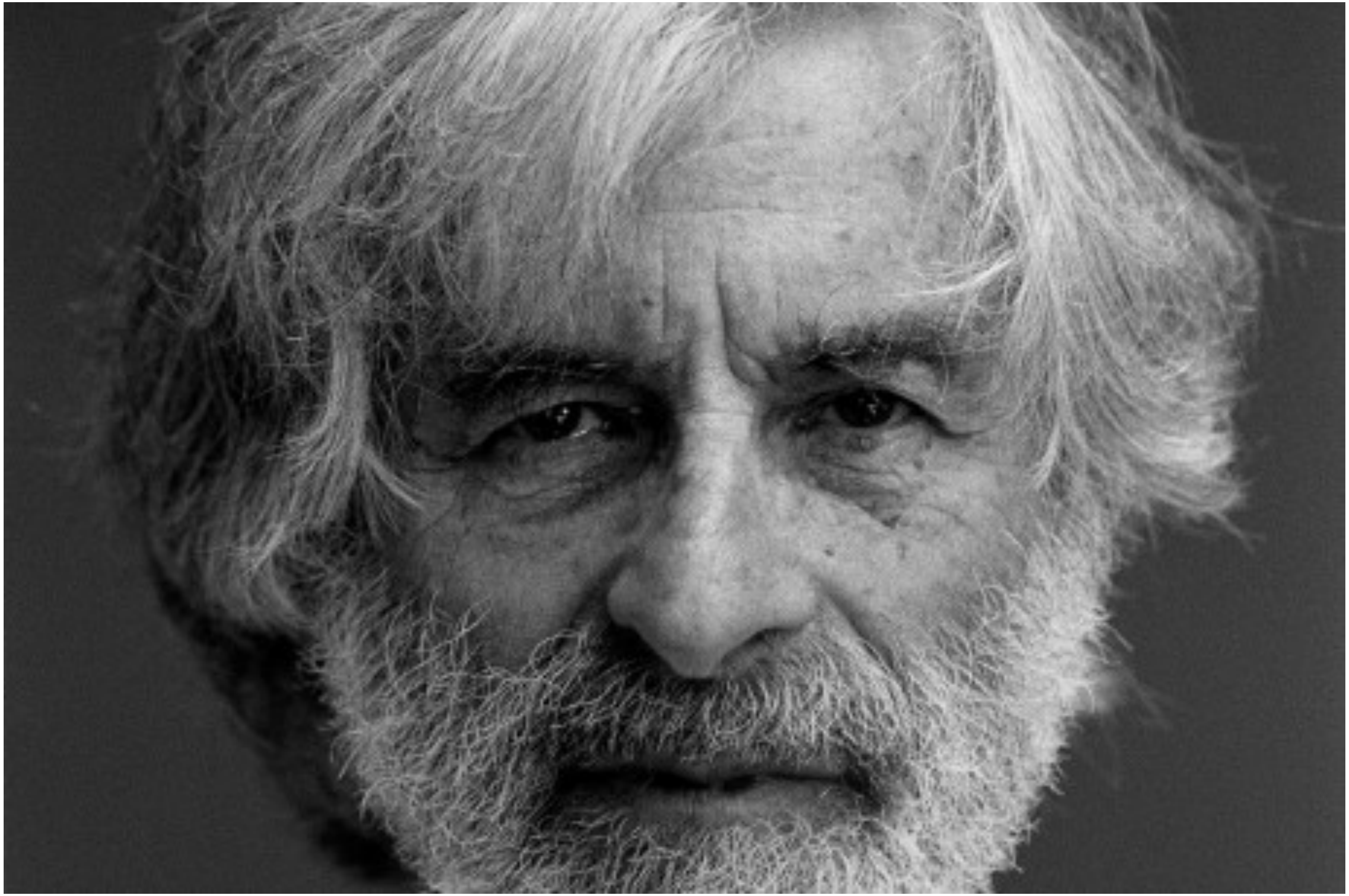
```
let f x = send x to h;
          r ← receive_from h
          return r
 in (f 42) + (f 239)
```

- Horizontal System Decomposition



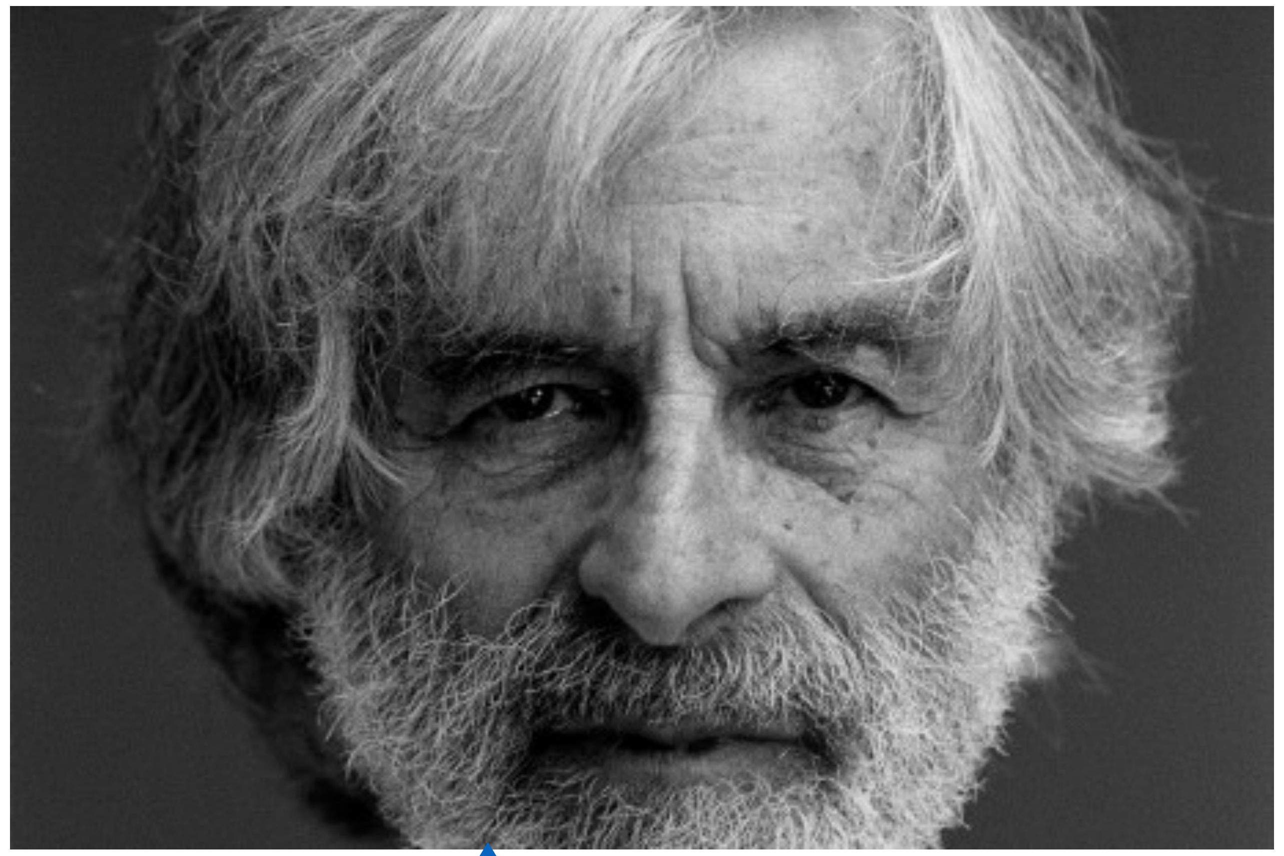
- Inter-Protocol Dependencies





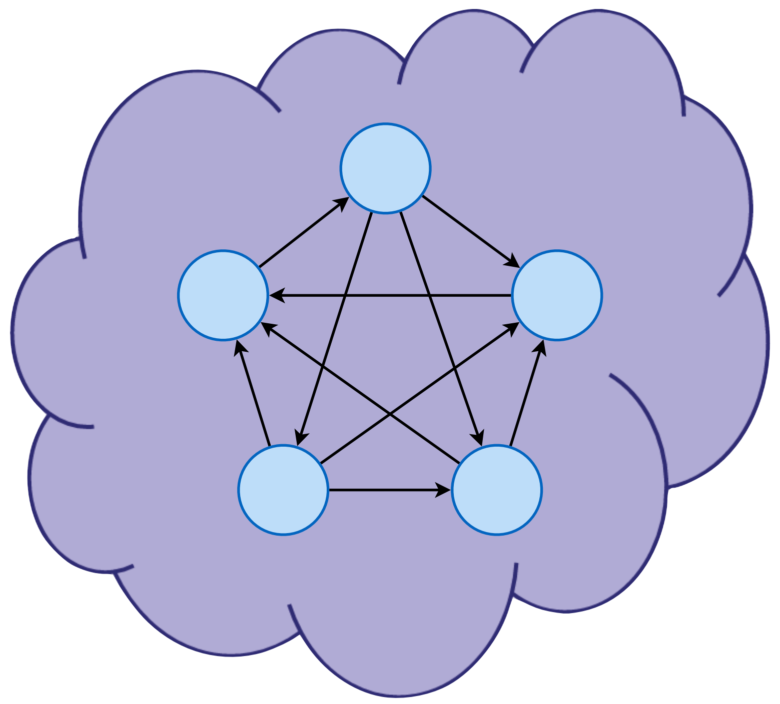
Composition: A way to make proofs harder

(Lamport, 1997)



When distracting *language features* are removed and the underlying *mathematics* is revealed, compositional reasoning is seen to be of little use.

"mathematics" "language features"

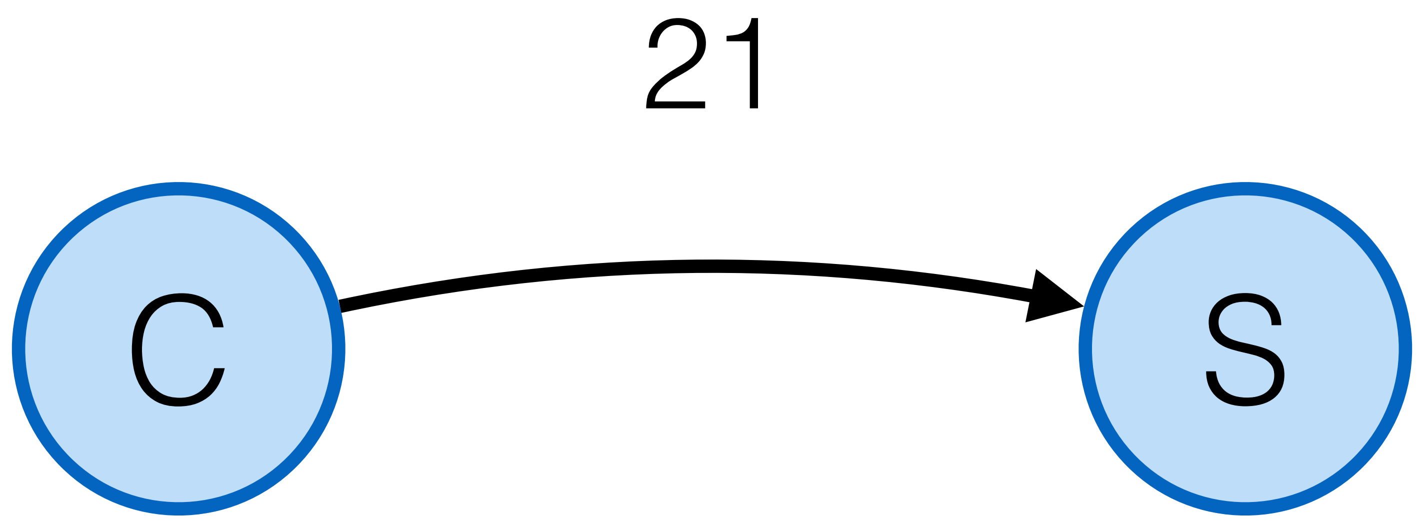
 $\vdash \{P\}\ c\ \{Q\}$

precondition postcondition

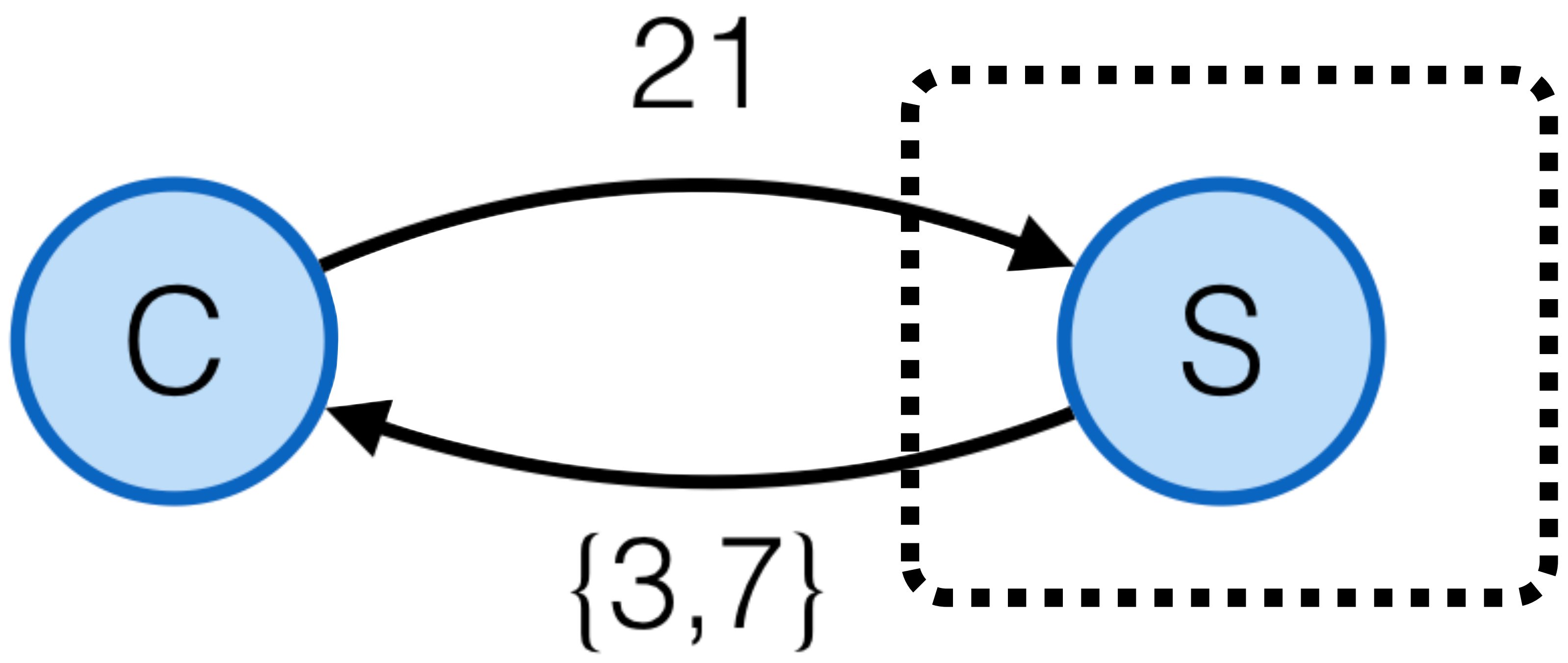
If the initial state satisfies P, then, after c terminates, the final state satisfies Q.

Working Example:
Cloud Compute System

Cloud Compute



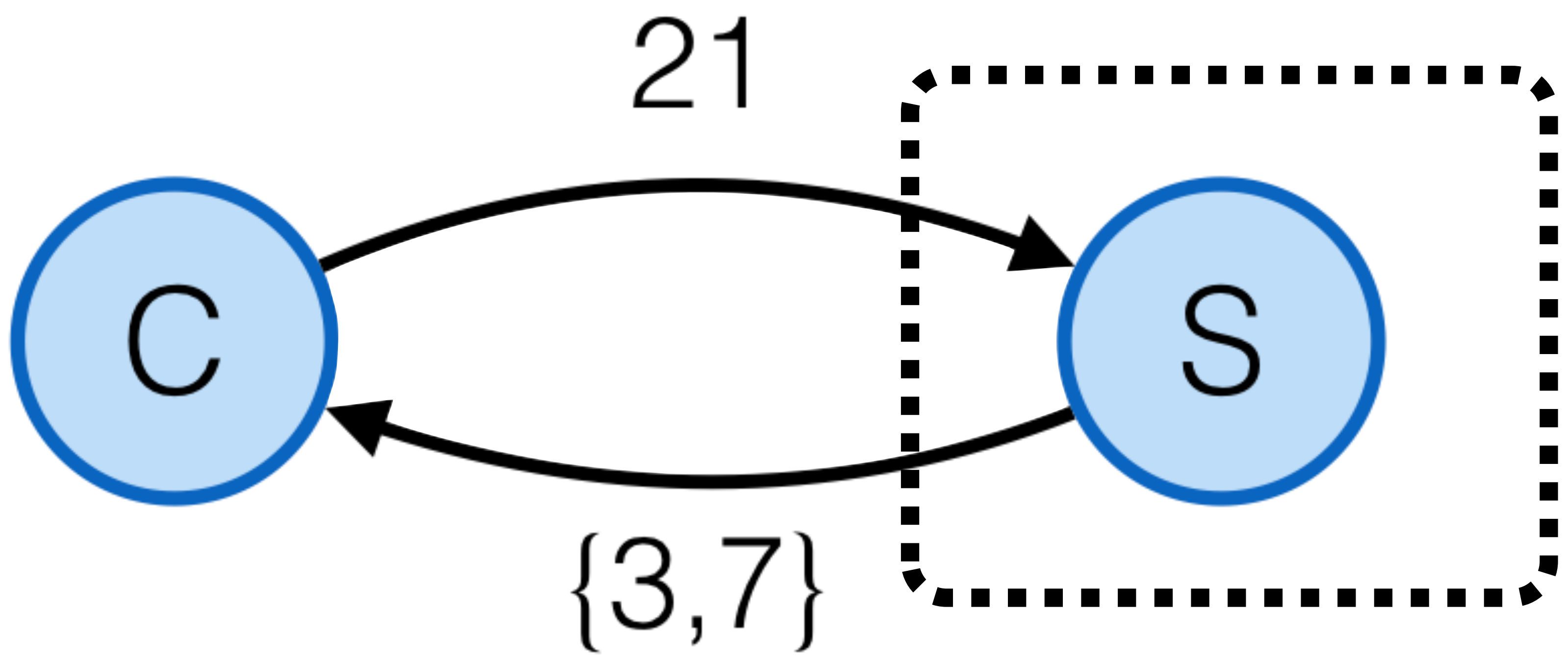
Cloud Compute: Server



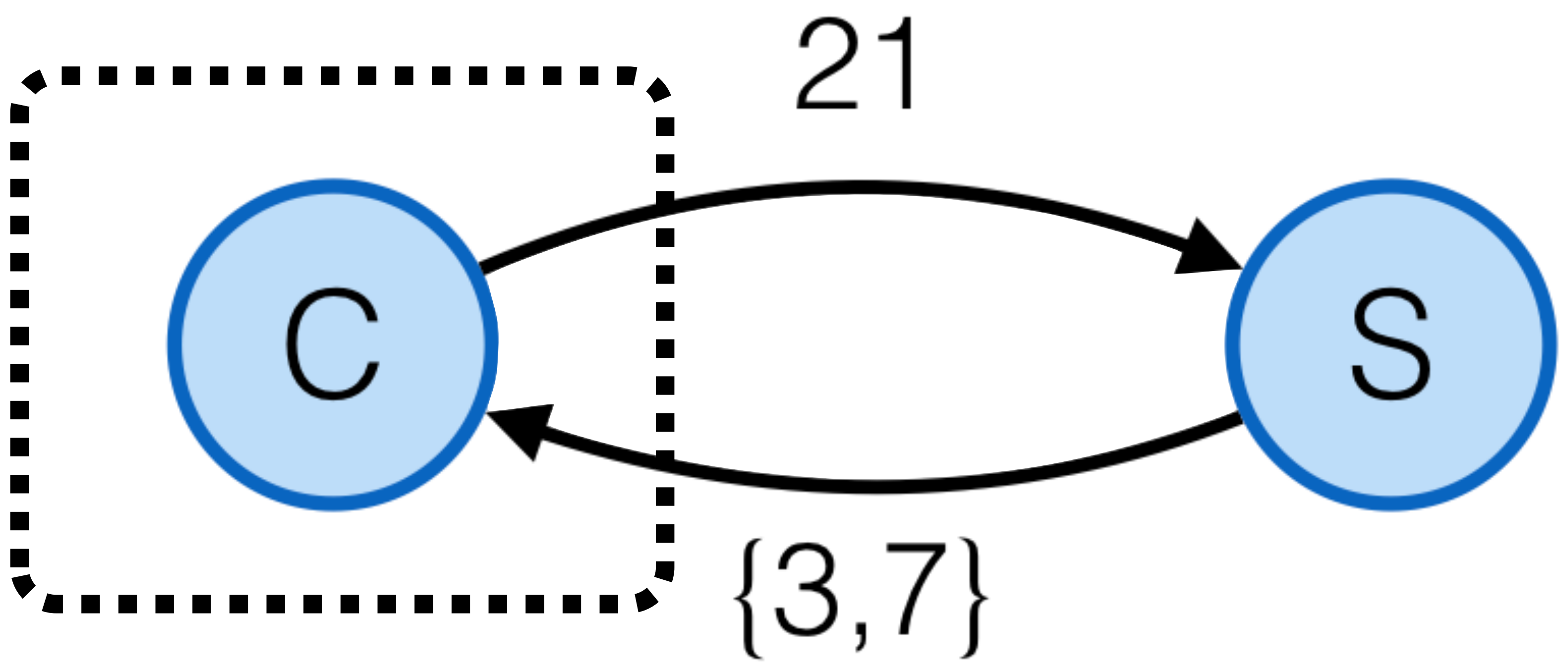
Cloud Compute: Server

```
while True:
  (from, n) <- recv
  send (n, factors(n)) to from
```

Cloud Compute: Server



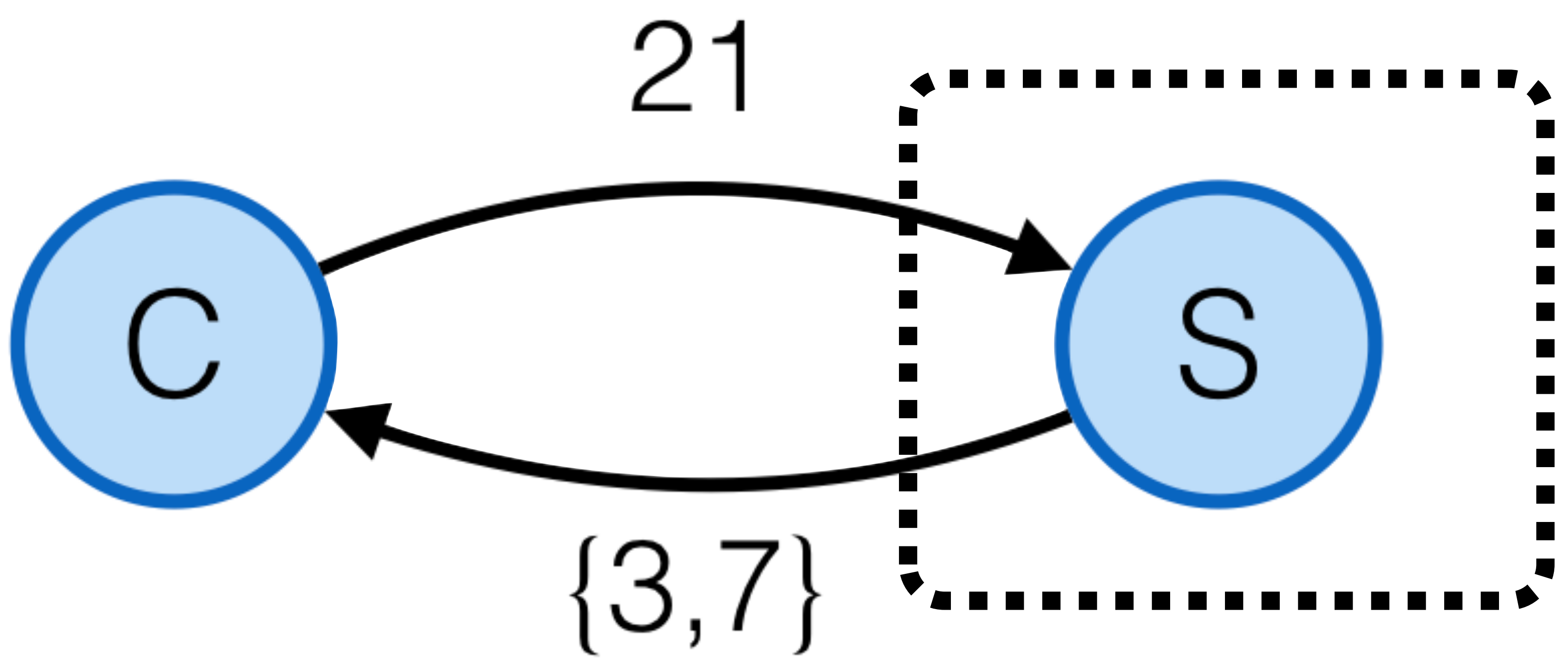
Cloud Compute: Client



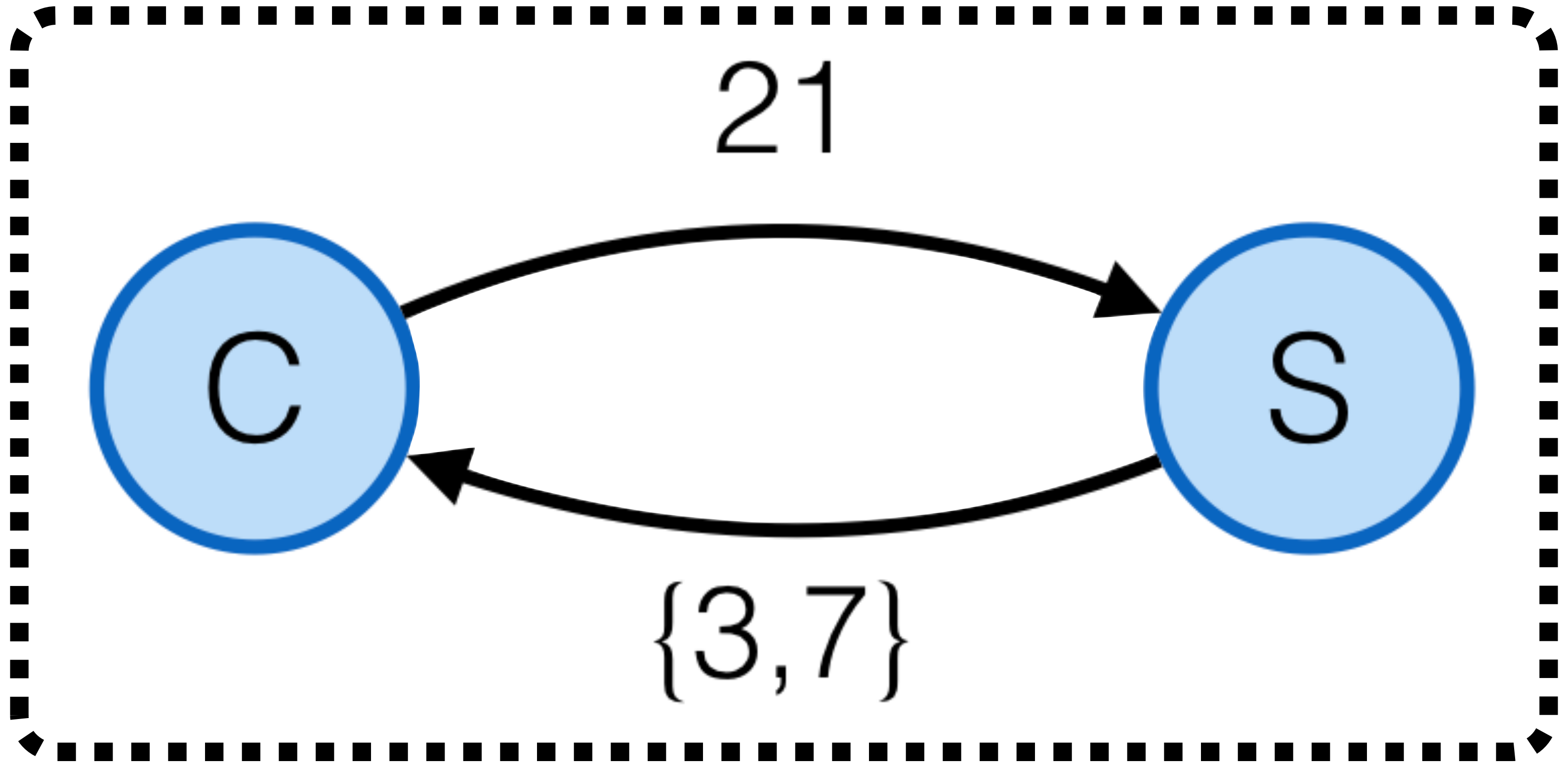
Cloud Compute: Client

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

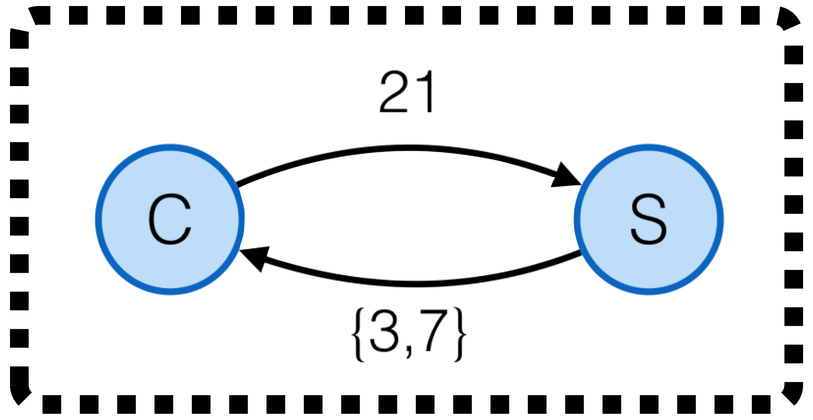
Protocols



Protocols



Protocols

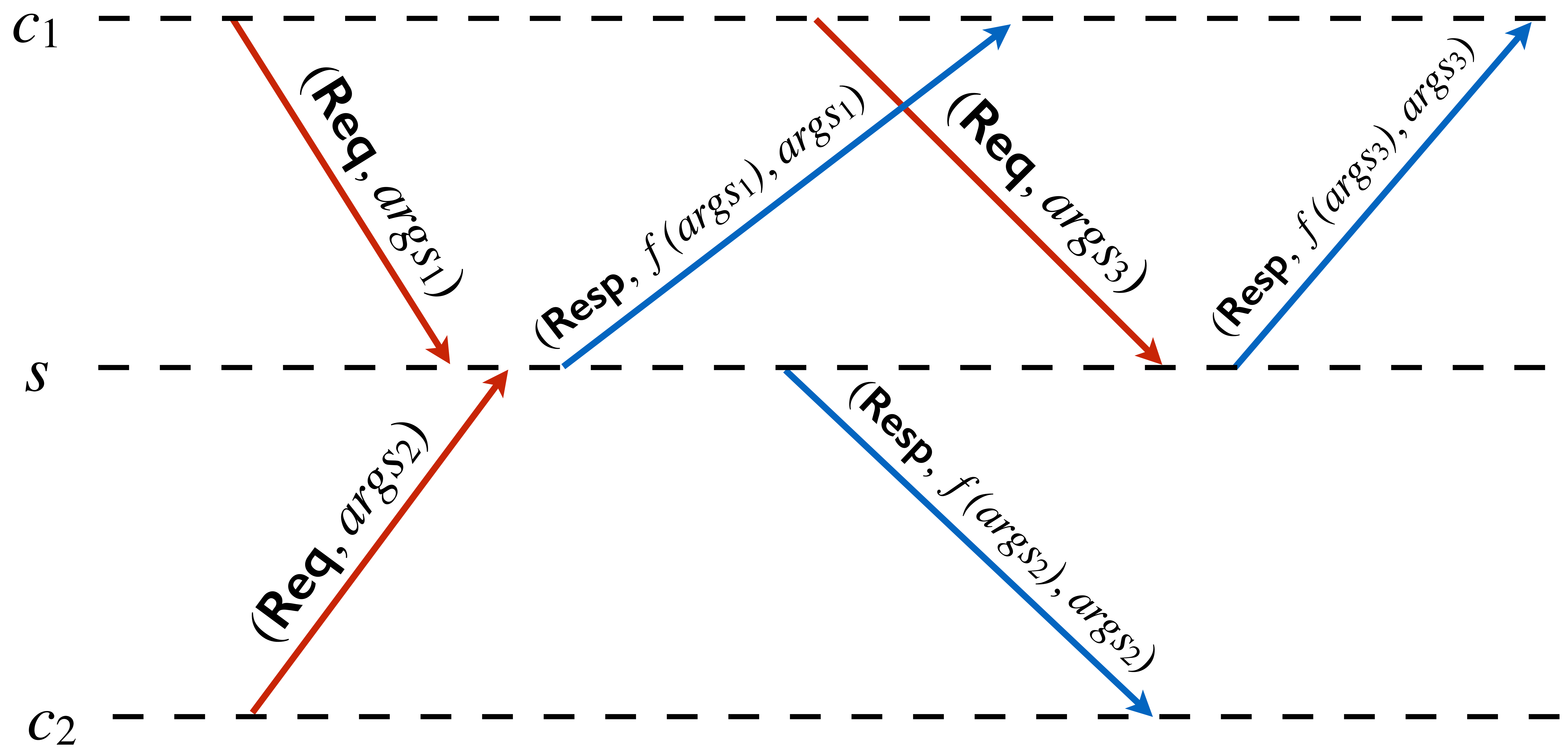


State:

abstract state of each node +
all *ever sent* messages

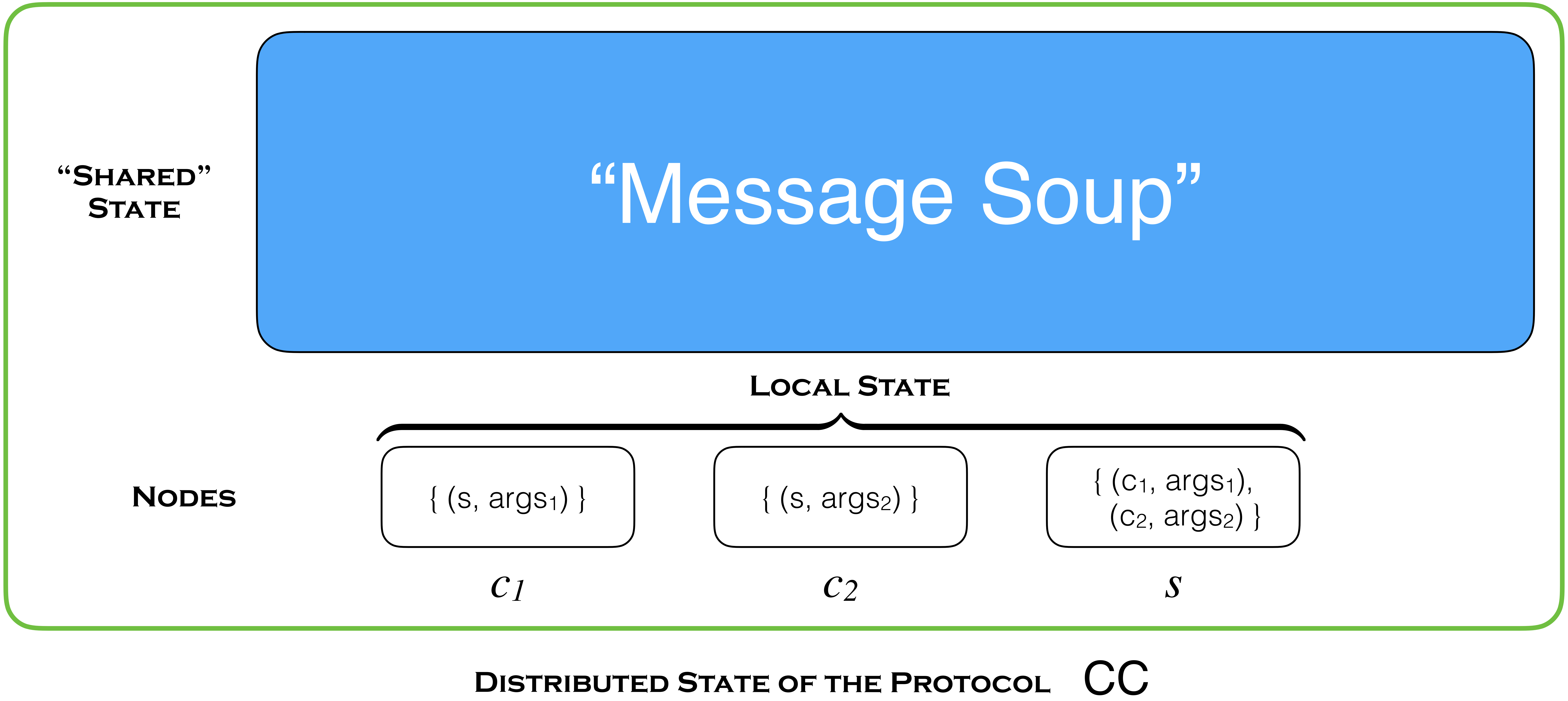
Transitions:

allowed sends and receives



c_1
s
c_2
$(\mathbf{Req}, args_1)$
$(\mathbf{Req}, args_2)$
$(\mathbf{Resp}, f(args_1), args_1)$
$(\mathbf{Req}, args_3)$
$(\mathbf{Resp}, f(args_2), args_2)$
$(\mathbf{Resp}, f(args_3), args_3)$

Cloud Compute State

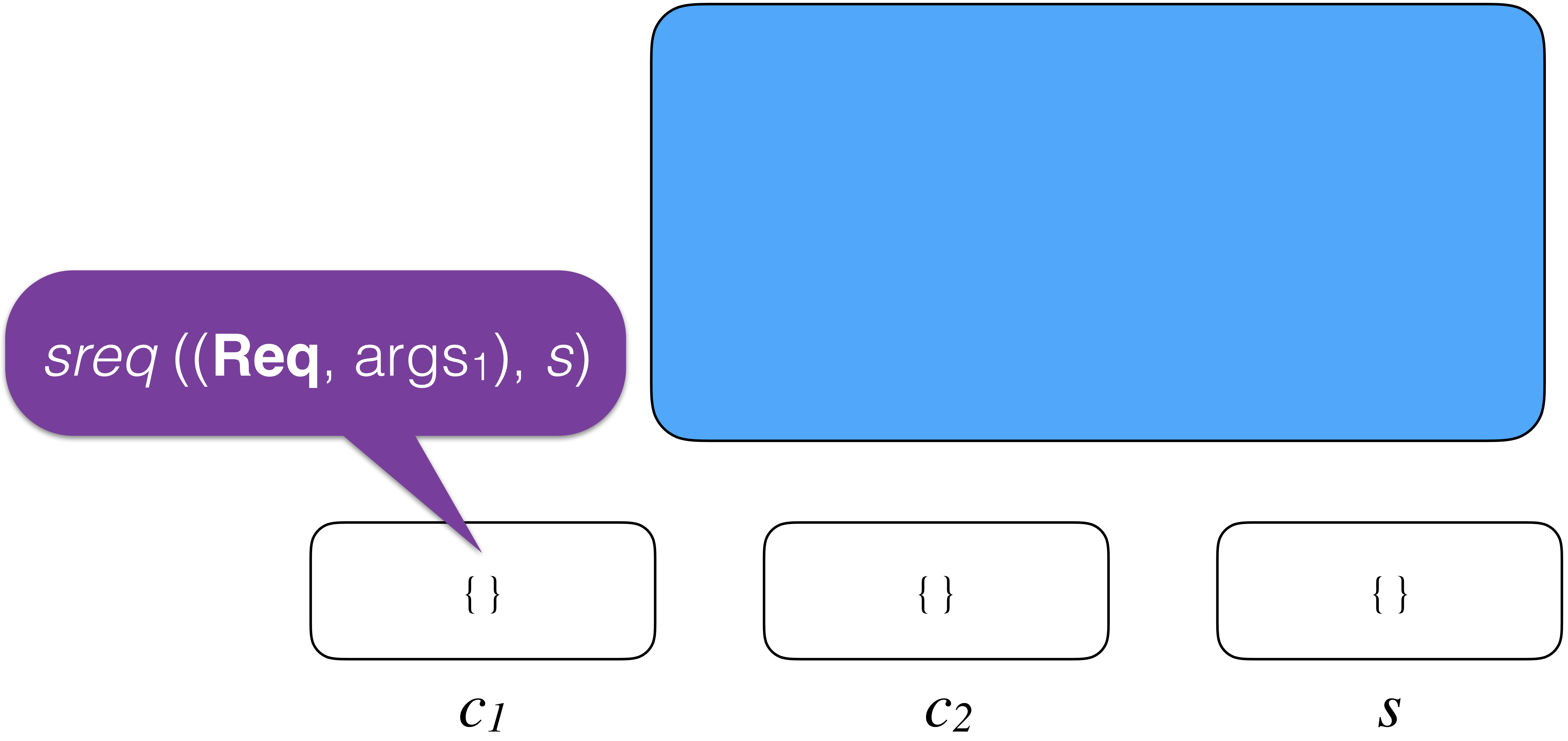


Send-Transitions

τ_s	Requires (m, to)	Ensures
$sreq$	$n \in \overline{C} \;\wedge\; to \in \overline{S} \;\wedge$ $n \mapsto rs \;\wedge\; m = (\mathsf{Req}, args) \;\wedge$ $args \;\in\; \mathsf{dom}(f)$	$n \mapsto (to, args) \uplus rs$
$sresp$	$n \in \overline{S} \;\wedge\; f(args) = v \;\wedge$ $n \mapsto (to, args) \uplus rs \;\wedge$ $m = (\mathsf{Resp}, v, args)$	$n \mapsto rs$

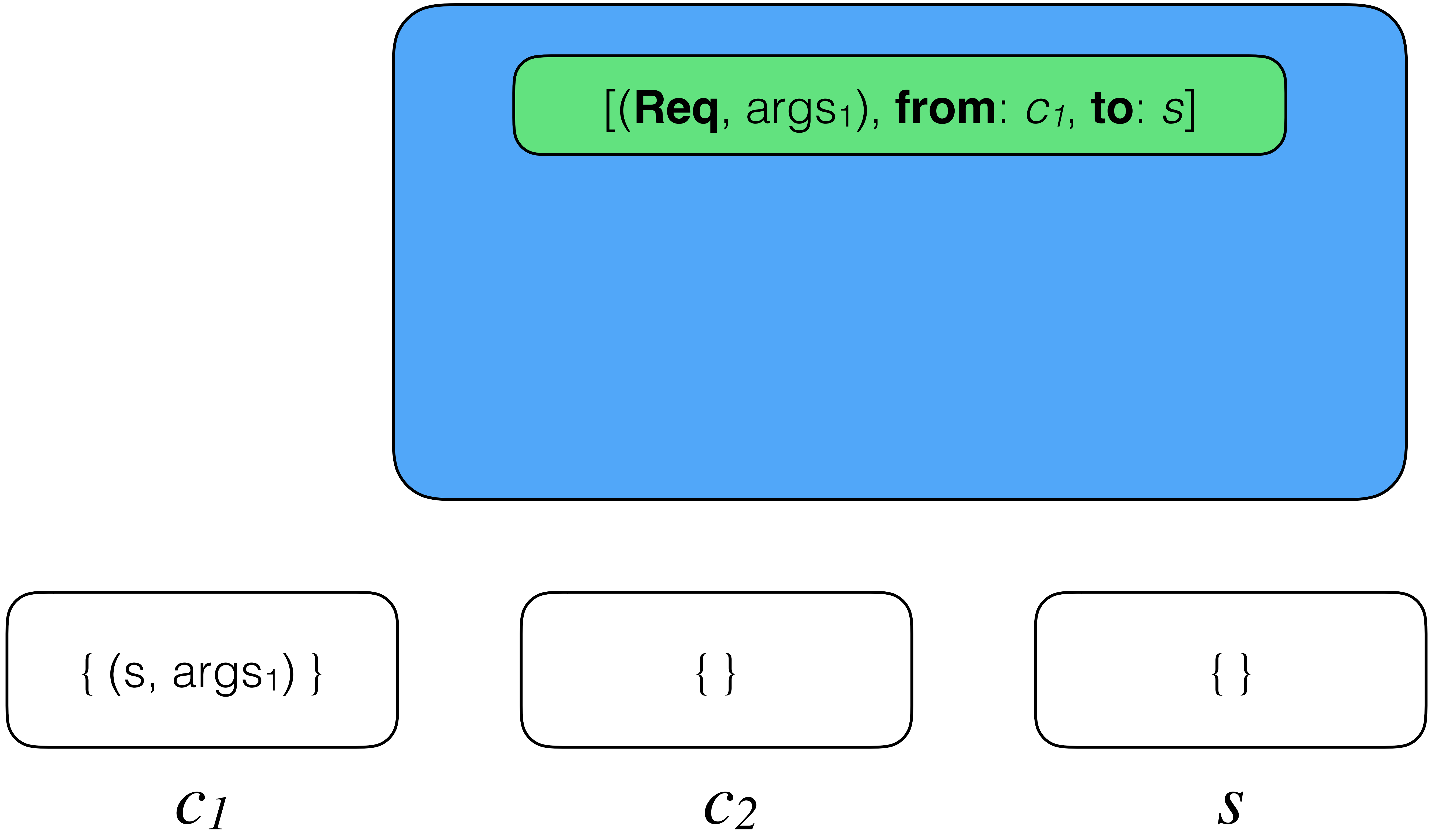
Send-Transitions

τ_s	Requires (m, to)	Ensures
$sreq$	$n \in \overline{C} \;\wedge\; to \in \overline{S} \;\wedge$ $n \mapsto rs \;\wedge\; m = (\mathsf{Req}, args) \;\wedge$ $args \;\in\; \mathsf{dom}(f)$	$n \mapsto (to, args) \uplus rs$
$sresp$	$n \in \overline{S} \;\wedge\; f(args) = v \;\wedge$ $n \mapsto (to, args) \uplus rs \;\wedge$ $m = (\mathsf{Resp}, v, args)$	$n \mapsto rs$



Send-Transitions

τ_s	Requires (m, to)	Ensures
$sreq$	$n \in \overline{C} \;\wedge\; to \in \overline{S} \;\wedge$ $n \mapsto rs \;\wedge\; m = (\mathsf{Req}, args) \;\wedge$ $args \;\in\; \mathsf{dom}(f)$	$\boxed{n \mapsto (to, args) \uplus rs}$
$sresp$	$n \in \overline{S} \;\wedge\; f(args) = v \;\wedge$ $n \mapsto (to, args) \uplus rs \;\wedge$ $m = (\mathsf{Resp}, v, args)$	$n \mapsto rs$

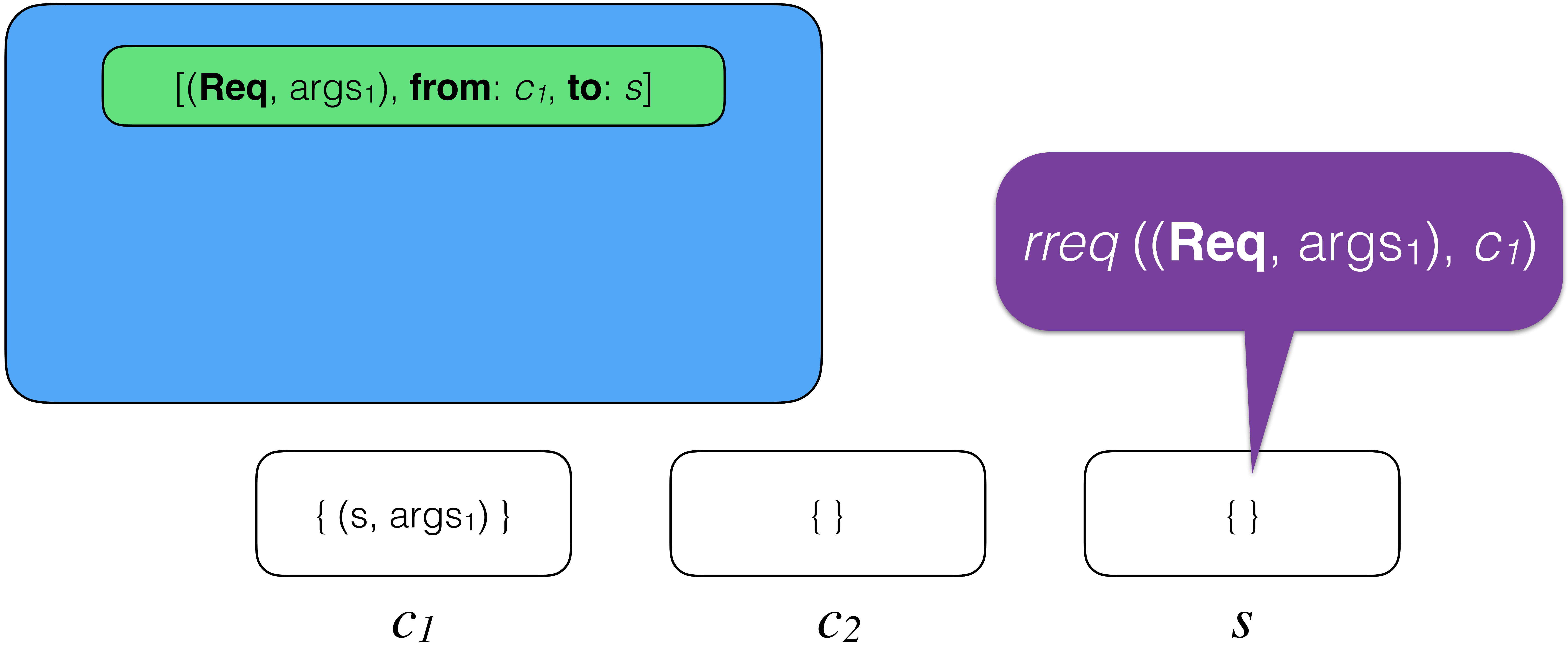


Receive-Transitions

τ_r	Requires (m, from)	Ensures
rreq	$n \in \overline{S}$ & $n \mapsto rs$ & $m = (\mathsf{Req}, \mathit{args})$	$n \mapsto (\mathit{from}, \mathit{args}) \cup rs$
rresp	$n \in \overline{C}$ & $n \mapsto (\mathit{from}, \mathit{args}) \cup rs$ & $m = (\mathsf{Resp}, \mathit{ans}, \mathit{args})$	$n \mapsto rs$

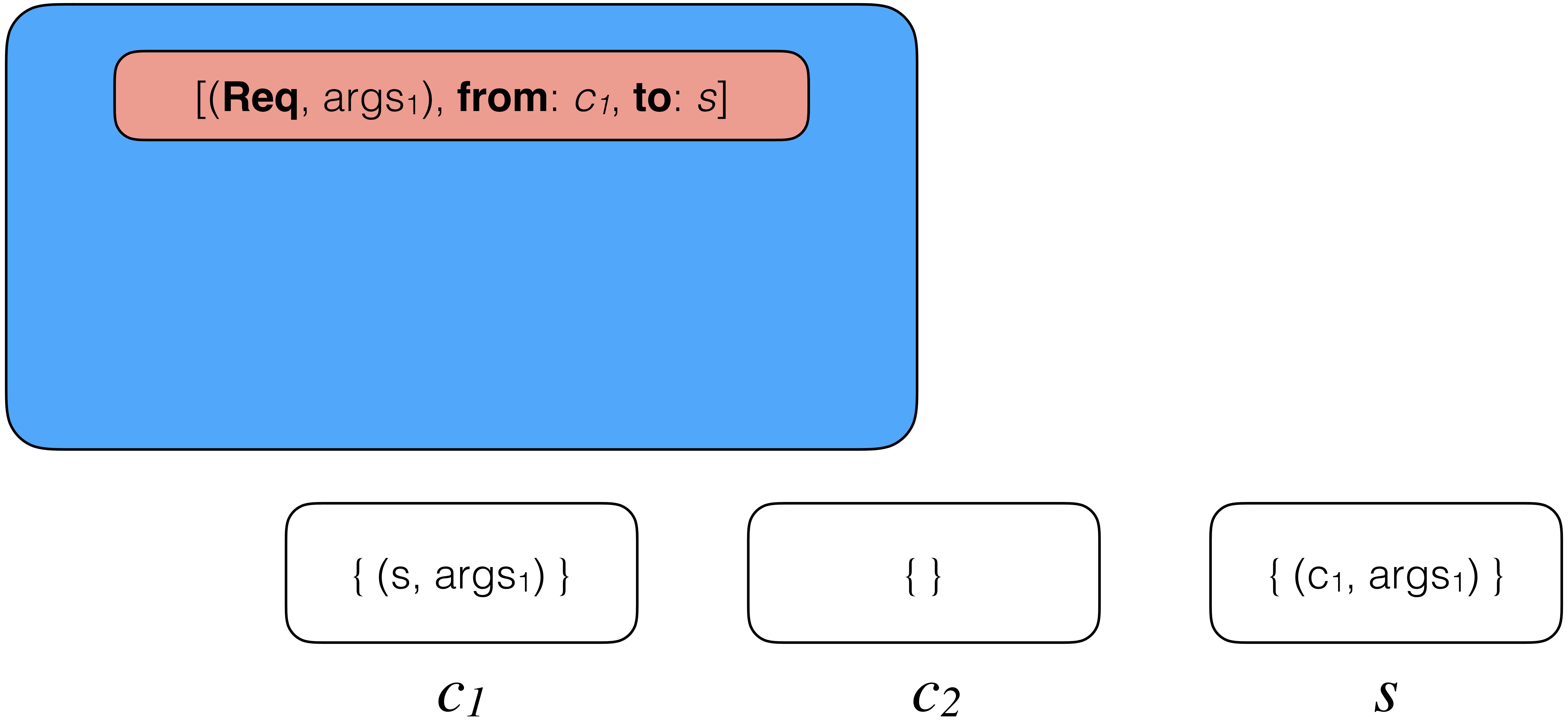
Receive-Transitions

τ_r	Requires (m, from)	Ensures
$rreq$	$n \in \overline{S}$ && $n \mapsto rs$ && $m = (\mathsf{Req}, args)$	$n \mapsto (\mathit{from}, args) \cup rs$
$rresp$	$n \in \overline{C}$ && $n \mapsto (\mathit{from}, args) \cup rs$ && $m = (\mathsf{Resp}, ans, args)$	$n \mapsto rs$



Receive-Transitions

τ_r	Requires (m, from)	Ensures
rreq	$n \in \overline{S}$ && $n \mapsto rs$ && $m = (\mathsf{Req}, \mathit{args})$	$\boxed{n \mapsto (\mathit{from}, \mathit{args}) \cup rs}$
rresp	$n \in \overline{C}$ && $n \mapsto (\mathit{from}, \mathit{args}) \cup rs$ && $m = (\mathsf{Resp}, \mathit{ans}, \mathit{args})$	$n \mapsto rs$

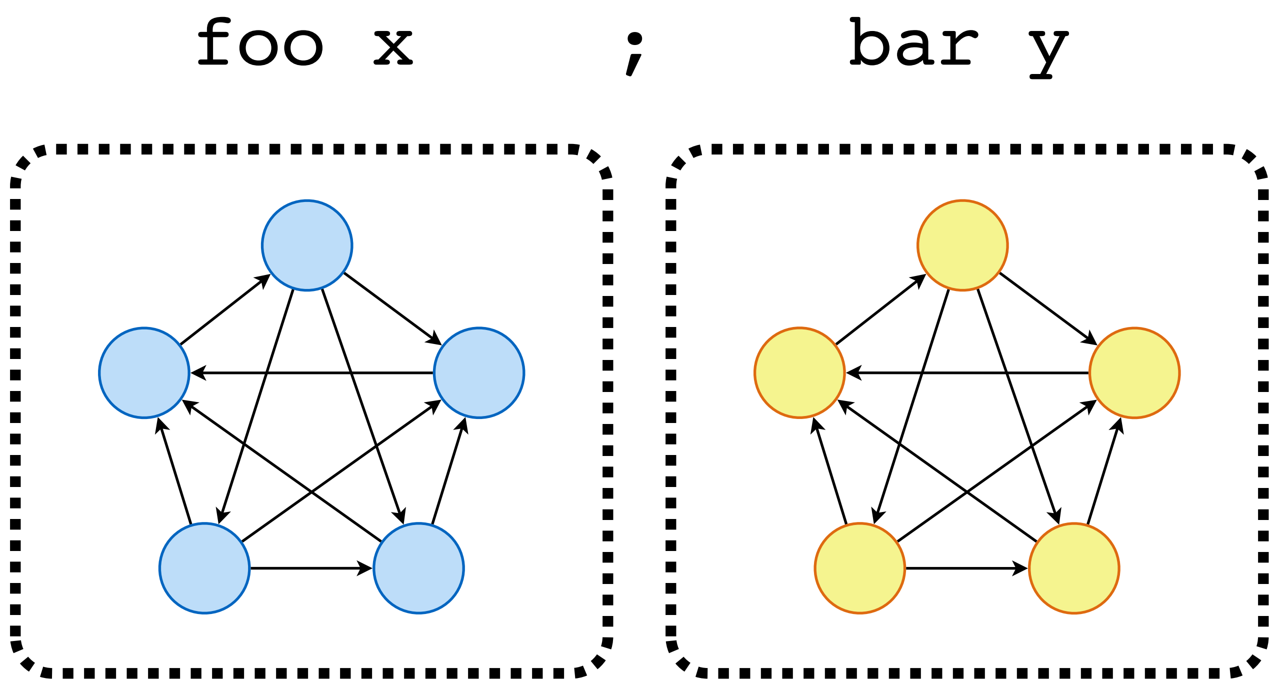


Composition in Distributed Systems

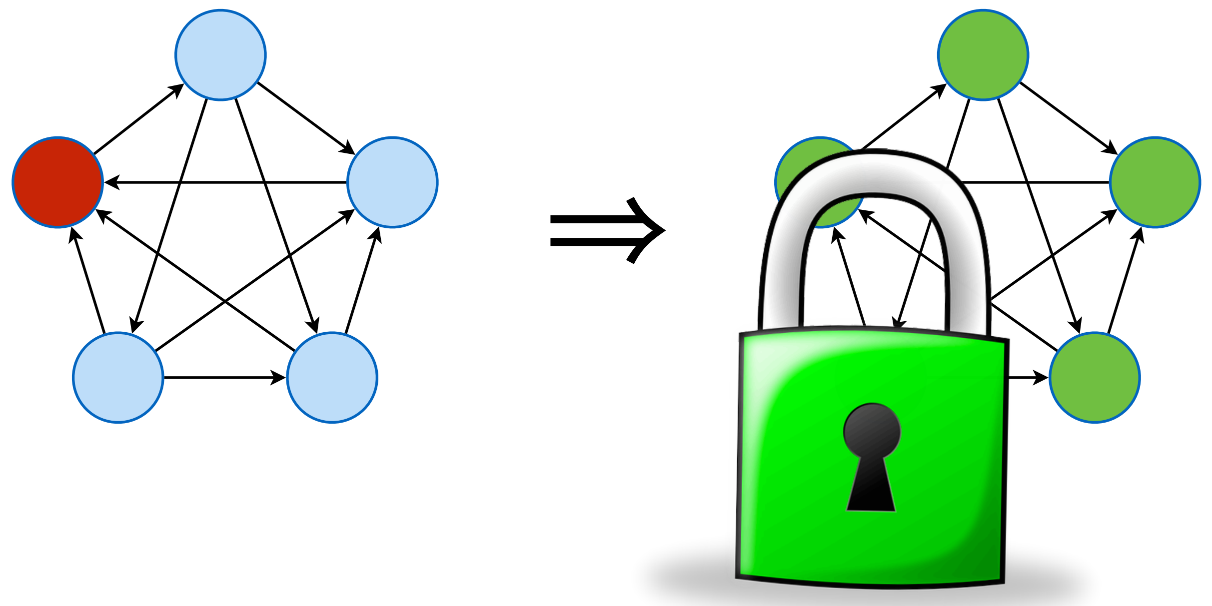
- Modular Program Verification

```
let f x = send x to h;
          r ← receive_from h
          return r
 in (f 42) + (f 239)
```

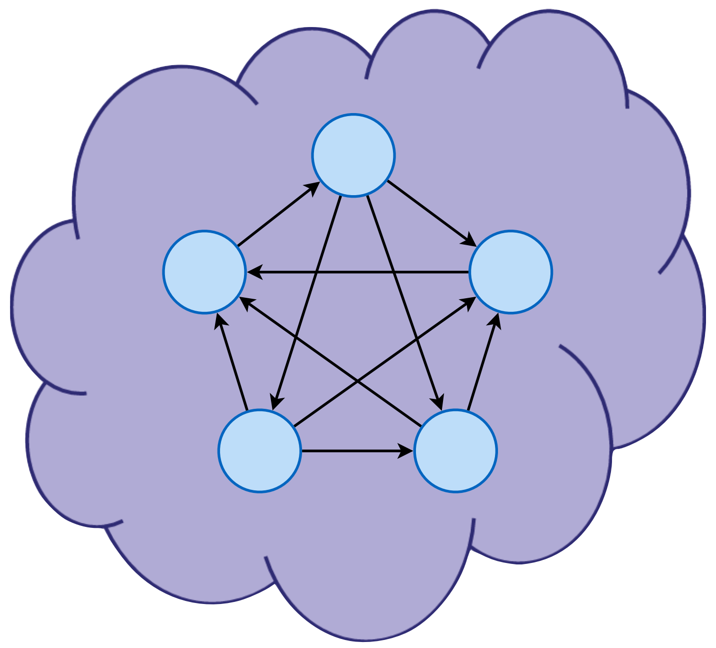
- Horizontal System Decomposition



- Inter-Protocol Dependencies

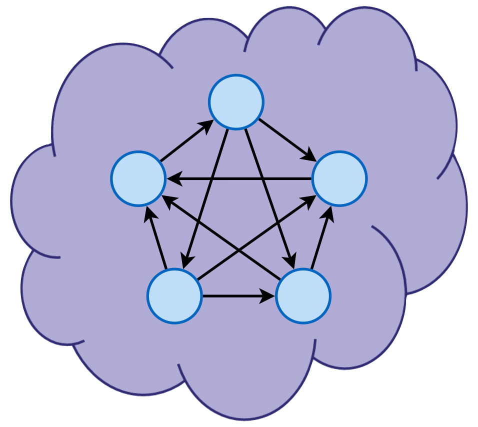


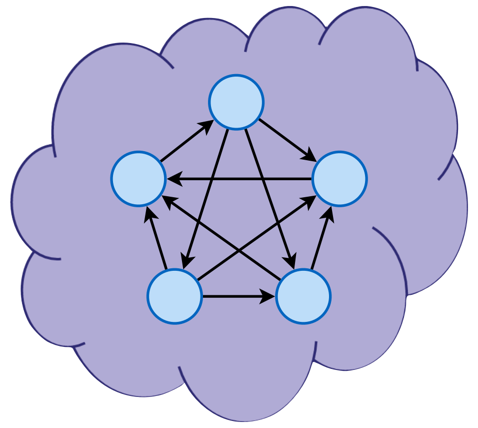
From Protocols to Hoare Specs



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

From Protocols to Hoare Specs

$\boxed{\mathrm{tr}} \in$  $\qquad P \Rightarrow Pre_{\boxed{\mathrm{tr}}}$

 $\vdash \{ \quad \}$ **send** m $\boxed{\mathrm{tr}}$ to h $\{ \qquad \}$

Don’t know if applicable!
(e.g., args = [0])

```
letrec server_loop _ =
  (from, args) ← blocking_receive();
  let ans = factor(args) in
  send_sresp ((Resp, ans, args), from);
  server_loop()
in server_loop()
```

Send-transitions

τ_s	Requires (m, to)	Ensures
$sreq$	$n \in \overline{C} \;\wedge\; to \in \overline{S} \;\wedge$ (a) $n \mapsto rs \;\wedge \boxed{m = (\mathsf{Req}, args)} \wedge$ $\boxed{args \;\in\; \mathsf{dom}(f)}$ (c)	$n \mapsto (to, args) \uplus rs$
$sresp$	$n \in \overline{S} \;\wedge\; f(args) = v \;\wedge$ $n \mapsto (to, args) \uplus rs \;\wedge$ $m = (\mathsf{Resp}, v, args)$	$n \mapsto rs$

Receive-transitions

τ_r	Requires $(m, from)$	Ensures
$rreq$	$n \in \overline{S}$ && $n \mapsto rs$ && $\boxed{m = (\mathsf{Req}, args)}$ (b)	$n \mapsto (from, args) \uplus rs$
$rresp$	$n \in \overline{C}$ && $n \mapsto (from, args) \uplus rs$ && $m = (\mathsf{Resp}, ans, args)$	$n \mapsto rs$

Inductive Invariant $\mathbb{Inv}_1$

$$\mathbb{Inv}_1(s) \triangleq \forall \mathbf{m} \in s.\mathrm{MS},\ \mathbf{m} = \langle \textit{from}, \textit{to}, -, (\mathbf{Req}, \textit{args})\rangle \Rightarrow \textit{args} \in \mathbf{dom}(\texttt{factor})$$

A rule for Invariant Strengthening

$\mathbb{I}\mathrm{nv}_1(s) \triangleq \forall m \in s.\mathsf{MS},\ m = \langle \mathit{from}, \mathit{to}, \text{-}, (\mathsf{Req}, \mathit{args})\rangle$
$\Rightarrow \mathit{args} \in \mathrm{dom}(\texttt{factor})$

$$\frac{\Gamma; \langle \ell \mapsto \mathcal{P}_\ell \uplus W, H\rangle \overset{n}{\vdash} c : \{P\}\{Q\} \qquad I \text{ is inductive } \mathit{wrt.}\ \mathcal{P}_\ell \qquad \mathcal{I} \triangleq \forall s, \mathsf{this}\ s \Rightarrow I(s)}{\Gamma; \langle \ell \mapsto \boxed{\mathsf{WithInv}(\mathcal{P}_\ell, I)} \uplus W, H\rangle \overset{n}{\vdash} c : \{P \wedge \mathcal{I}\}\{Q \wedge \mathcal{I}\}}$$

A "protocol combinator"

$\mathbb{Inv}_1(s) \triangleq \forall m \in s.\mathsf{MS},\ m = \langle \mathit{from}, \mathit{to}, \text{-}, (\mathsf{Req}, \mathit{args})\rangle$
$\Rightarrow \mathit{args} \in \mathrm{dom}(\mathtt{factor})$

WithInv(CC, $\mathbb{Inv}_1$)

$\mathsf{CCI}_1 \vdash$
```
letrec server_loop _ =
    (from, args) ← blocking_receive();
    let ans = factor(args) in
    send_sresp ((Resp, ans, args), from);
    server_loop()
  in server_loop():
```

{ this *is a server* $\wedge\ \exists rs,$ this $\rightarrowtail rs$ }

{ False }

More Implementations for Cheap

A Batching Server

$$
\begin{array}{l}
\mathbf{letrec}\ \textsf{receive\_batch}\ (k : \textsf{nat}) \triangleq \\
\quad \mathbf{if}\ k = k' + 1 \\
\quad \mathbf{then}\ \mathit{fargs} \leftarrow \textsf{receive\_req}\ (); \\
\qquad\quad \mathit{rest} \leftarrow \textsf{receive\_batch}\ k'; \\
\qquad\quad \mathbf{return}\ \mathit{fargs} :: \mathit{rest} \\
\quad \mathbf{else\ return}\ [\,]
\end{array}
$$

$$
\begin{array}{l}
\mathbf{letrec}\ \textsf{send\_batch}\ (\mathit{rs} : [(\textsf{Node}, [\textsf{nat}])]) \triangleq \\
\quad \mathbf{if}\ \mathit{rs} = (\mathit{from}, \mathit{args}) :: \mathit{rs}' \\
\quad \mathbf{then\ let}\ v = f(\mathit{args})\ \mathbf{in} \\
\qquad \textsf{send}[\mathit{sresp}, \ell]((\textsf{Resp}, v, \mathit{args}), \mathit{from}); \\
\qquad \textsf{send\_batch}\ \mathit{rs}' \\
\quad \mathbf{else\ return}\ ()
\end{array}
$$

$$
\begin{array}{l}
\mathbf{letrec}\ \textsf{batch\_server}\ (\mathit{bsize} : \mathit{nat}) \triangleq \\
\qquad \mathit{reqs} \leftarrow \textsf{receive\_batch}\ \mathit{bsize};\ \textsf{send\_batch}(\mathit{reqs});\ \textsf{batch\_server}\ \mathit{bsize}
\end{array}
$$

$\mathsf{CCI}_1 \vdash \texttt{batch\_server(5)} :$

$\{\ \texttt{this}\ \textit{is a server} \wedge \exists \mathsf{rs}, \texttt{this} \rightarrowtail \mathsf{rs}\ \}$

$\{\ \mathsf{False}\ \}$

A Memoising Server

$$
\begin{array}{l}
\textbf{letrec}\ \mathsf{memo\_server}\ (mmap : \mathsf{map}) \triangleq \\
\quad (from, args) \leftarrow \mathsf{receive\_req}\ (); \\
\quad \textbf{let}\ ans = \mathsf{lookup}\ mmap\ \textbf{in} \\
\quad \textbf{if}\ ans \neq \bot\ \textbf{then}\ \mathsf{send}[sresp, \ell]((\mathsf{Resp}, ans, args), from); \\
\qquad\qquad\qquad\qquad \mathsf{memo\_server}\ mmap \\
\quad \textbf{else let}\ ans = f(args)\ \textbf{in} \\
\qquad\quad \mathsf{send}[sresp, \ell](m, (\mathsf{Resp}, ans, args)); \\
\qquad\quad \textbf{let}\ mmap' = \mathsf{update}(mmap, args, ans)\ \textbf{in} \\
\qquad\quad \mathsf{memo\_server}\ mmap'
\end{array}
$$

$\mathrm{CCI}_1 \vdash$ `memo_server({}) :`

{ `this` *is a server* ∧ ∃rs, `this` ↣ rs }

{ False }

A Client Implementation

$\mathrm{CCI}_1 \vdash$
```
fun compute_factor (arg, serv) =
    send_sreq ((Req, args), serv);
    r ← receive_resp();
    return r :
```

$\{$ `serv` *is a server* $\wedge$
`arg` $\in$ dom(`factor`) $\wedge$
`this` $\rightarrowtail \emptyset$ $\}$

$\{$ `res` $=$ `factor`(`arg`) $\wedge$ `this` $\rightarrowtail \emptyset$ $\}$

Cannot conclude `res = factor(args)`.

$$\mathsf{CCI}_1 \vdash \texttt{receive\_resp():}$$

$$\{\, \texttt{this} \mapsto \{\boxed{(\texttt{serv, arg})}\} \,\}$$

$$\{\, \texttt{<serv}, \texttt{this}, \bullet, \boxed{(\textbf{Resp}, \texttt{res}, \texttt{arg})}\texttt{>} \in \mathsf{MS} \;\wedge$$

$$\texttt{this} \mapsto \varnothing \,\}$$

Send-transitions

τ_s	Requires (m, to)	Ensures
$sreq$	$n \in \overline{C} \;\wedge\; to \in \overline{S} \;\wedge$ $n \mapsto rs \;\wedge\; m = (\mathsf{Req}, args) \;\wedge$ $args \;\in\; \mathsf{dom}(f)$	$n \mapsto (to, args) \uplus rs$
$sresp$	$n \in \overline{S} \;\wedge \boxed{f(args) = v} \wedge$ (b) $n \mapsto (to, args) \uplus rs \;\wedge$ $m = \boxed{(\mathsf{Resp}, v, args)}$ (a)	$n \mapsto rs$

Receive-transitions

τ_r	Requires $(m, from)$	Ensures
$rreq$	$n \in \overline{S}$ && $n \mapsto rs$ && $m = (\mathsf{Req}, args)$	$n \mapsto (from, args) \uplus rs$
$rresp$	$n \in \overline{C}$ && $n \mapsto (from, args) \uplus rs$ && $m = \boxed{(\mathsf{Resp}, ans, args)}$ (c)	$n \mapsto rs$

Inductive Invariant $\mathbb{I}\mathrm{nv}_2$

$$\mathbb{I}\mathrm{nv}_2(s) \triangleq \forall m \in s.\mathrm{MS},\ m = \langle -, -, -, (\mathbf{Resp}, \mathit{ans}, \mathit{args})\rangle$$
$$\Rightarrow \mathtt{factor}(\mathit{args}) = \mathit{ans})$$

$\mathbb{I}\mathrm{nv}_2(s) \triangleq \forall m \in s.\mathrm{MS},\ m = \langle -, -, -, (\mathrm{Resp}, \mathit{ans}, \mathit{args})\rangle$
$\Rightarrow \texttt{factor}(\mathit{args}) = \mathit{ans})$

$\mathrm{WithInv}(\mathrm{CCI}_1, \mathbb{I}\mathrm{nv}_2)$

$\mathrm{CCI}_2 \vdash$

```
fun compute_factor (arg, serv) =
    send_sreq ((Req, args), serv);
    r ← receive_resp();
    return r
```

$: \{\ \texttt{serv}\ \textit{is a server} \wedge$
$\quad \texttt{arg} \in \mathrm{dom}(\texttt{factor}) \wedge$
$\quad \texttt{this} \rightarrowtail \varnothing\ \}$

$\{\ \boxed{\texttt{res} = \texttt{factor}(\texttt{arg})} \wedge \texttt{this} \rightarrowtail \varnothing\ \}$

Composition in Distributed Systems

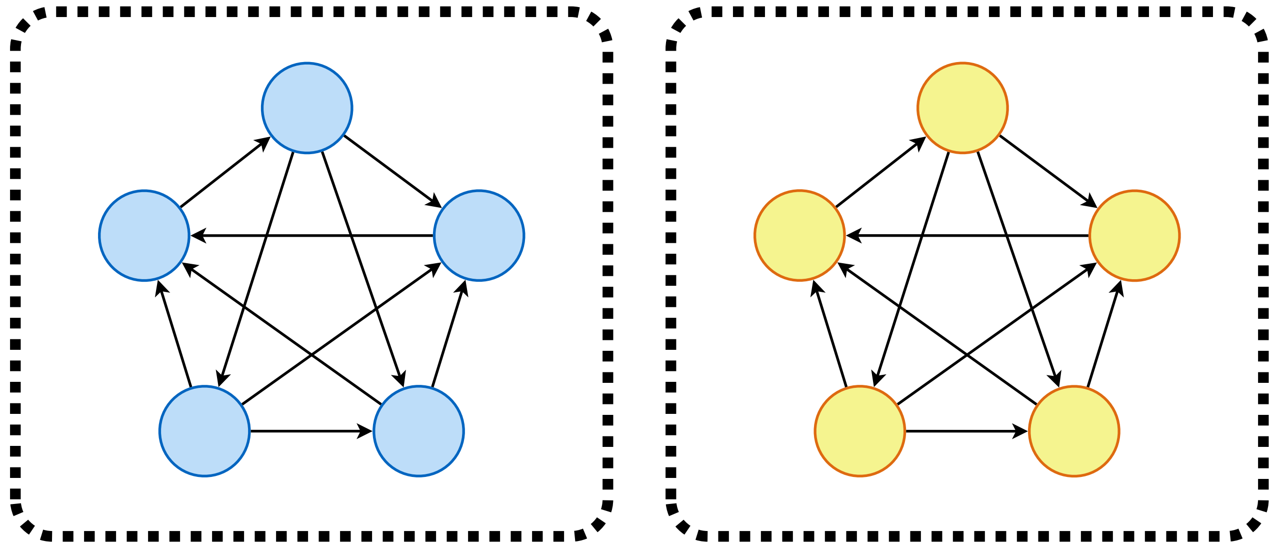
- Modular Program Verification

```
let f x = send x to h;
          r ← receive_from h
          return r
 in (f 42) + (f 239)
```

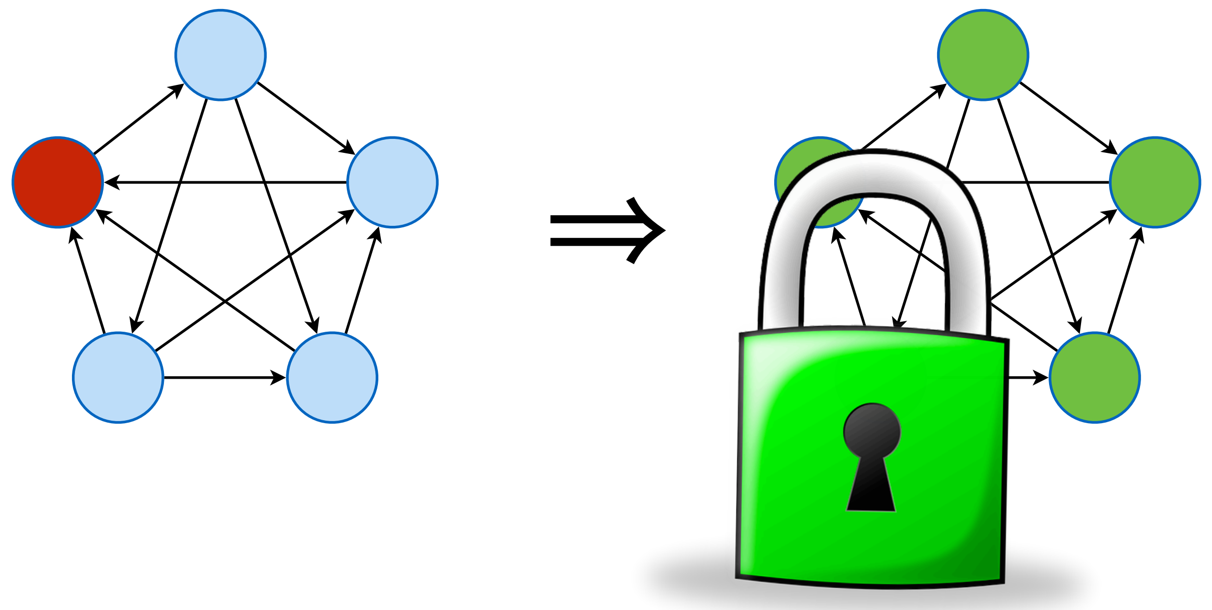
Protocol-aware logic +
Rule for *inductive invariants*

- Horizontal System Decomposition

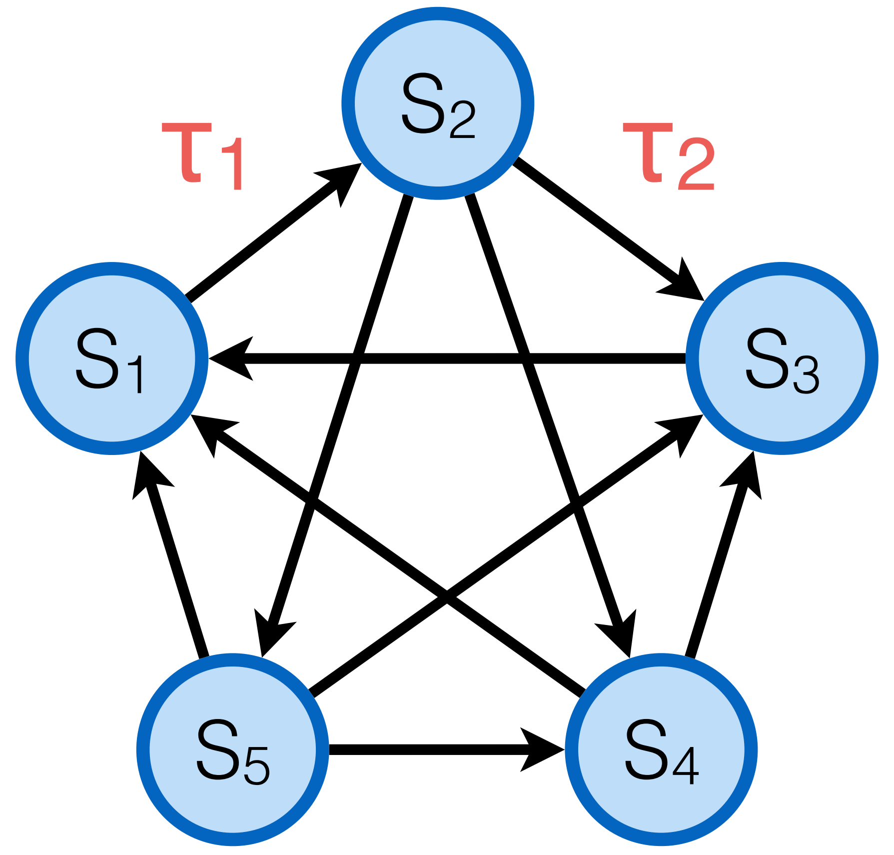
`foo x ; bar y`



- Inter-Protocol Dependencies



Horizontal System Decomposition



$\vdash$

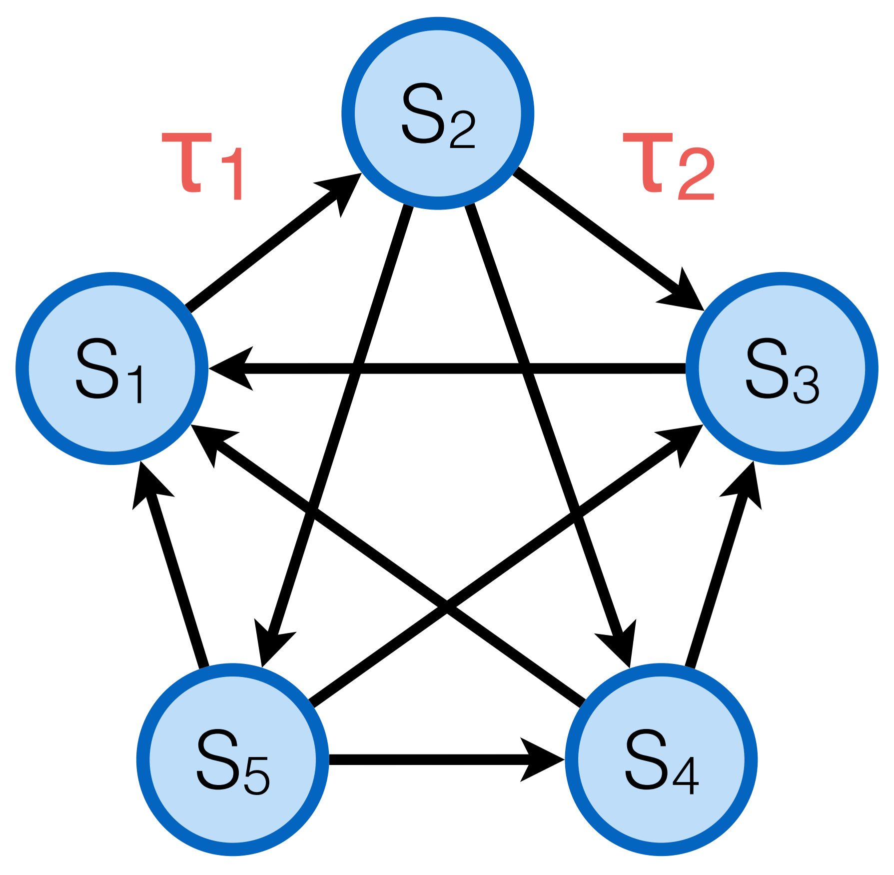
$\{s \mid s = S_1\}$

$\Big($ `send(msg);` `doStuff();` $\Big)\ \tau_1$

$\{s \mid s = S_2\}$

$\Big($ `m <- receive(c);` `doMoreStuff();` $\Big)\ \tau_2$

$\{s \mid s = S_3\}$

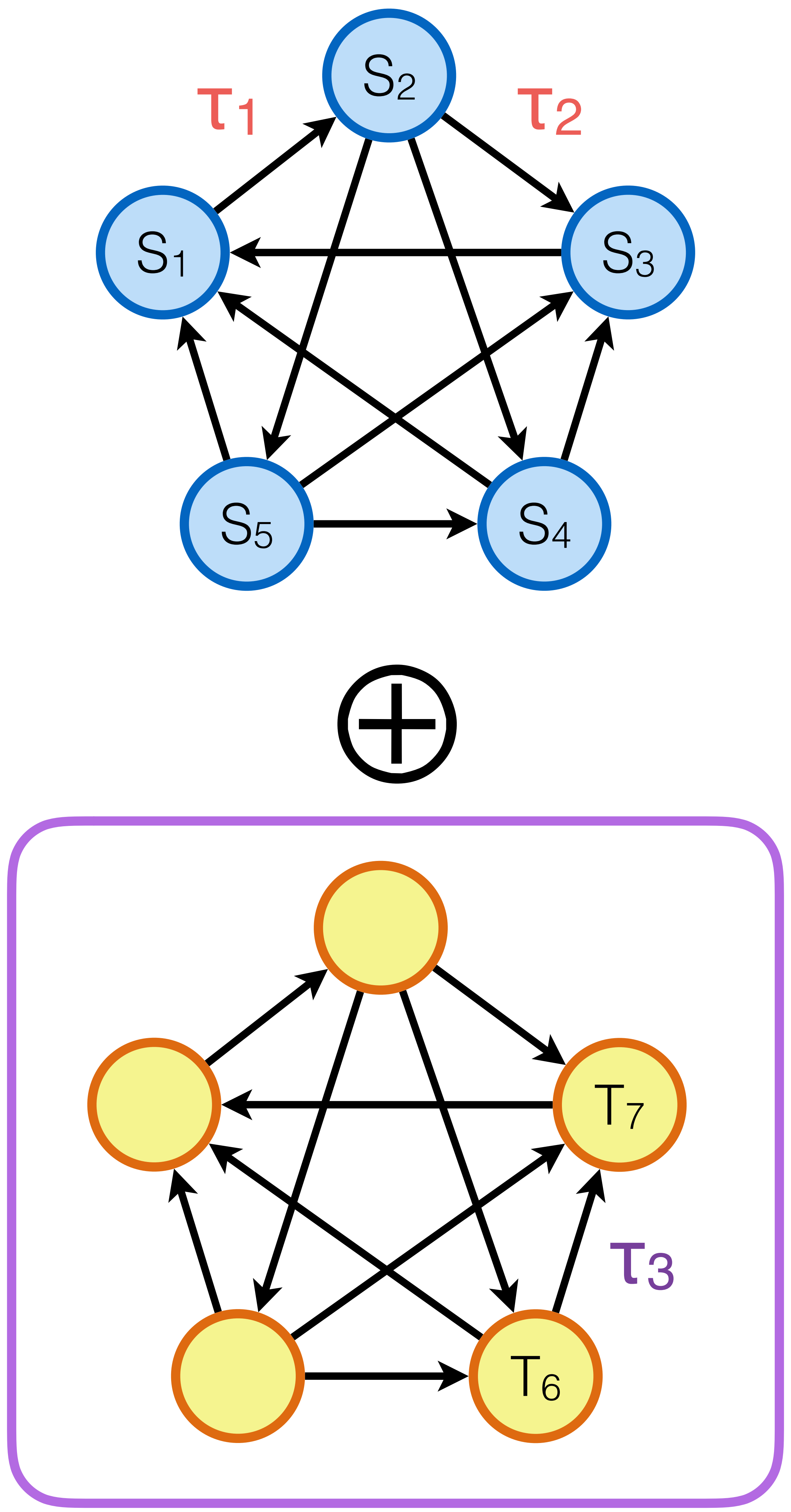


$\vdash$

```
{s | s = S1}
send(msg);
doStuff();

{s | s = S2}

m <- receive(c);
doMoreStuff();
{s | s = S3}

doOtherStuff();
{s | s = S3}
```

τ_1

τ_2



“frame”

$\vdash$

$\{s \mid s = S_1 \oplus T_6\}$

```
send(msg);
doStuff();
```

τ_1

$\{s \mid s = S_2 \oplus T_6\}$

```
m <- receive(c);
doMoreStuff();
```

τ_2

$\{s \mid s = S_3 \oplus T_6\}$

```
doOtherStuff();
```

$\{s \mid s = S_3 \oplus T_7\}$

τ_3

A Delegating Server

$\boxed{\mathrm{CCI}_1} \oplus \boxed{\mathrm{CCI}_2} \vdash$

```
letrec delegating_server (n′: Node) =
  (from, args) ← blocking receive();
  let ans = compute_factor(args) in
  send_sresp ((Resp, ans, args), from);
  delegating_server (n′)
in delegating_server(server)
```

Composition in Distributed Systems

- Modular Program Verification

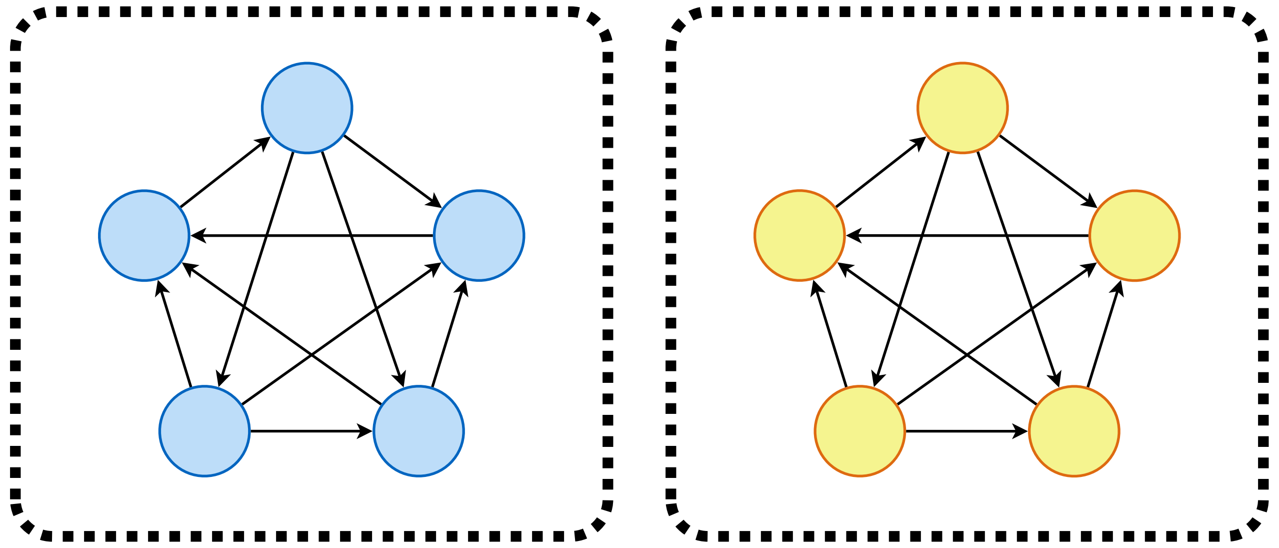
✓ **Protocol-aware logic +**
Rule for *inductive invariants*

```
let f x = send x to h;
          r ← receive_from h
          return r
 in (f 42) + (f 239)
```

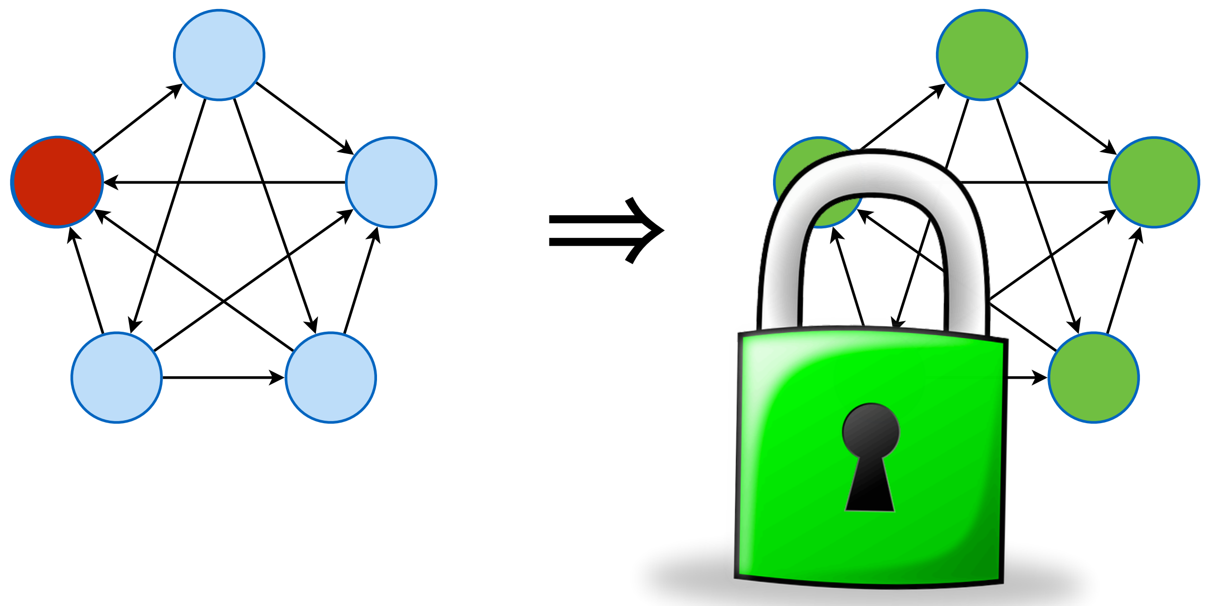
- Horizontal System Decomposition

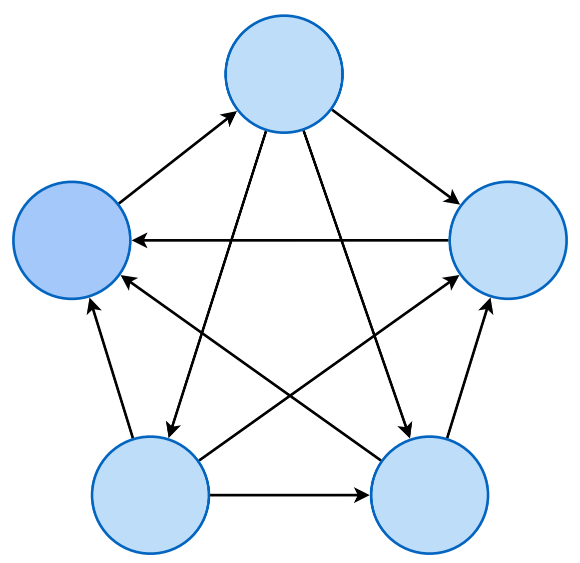
✓ **Framing wrt. a protocol**

```
foo x    ;    bar y
```



- Inter-Protocol Dependencies





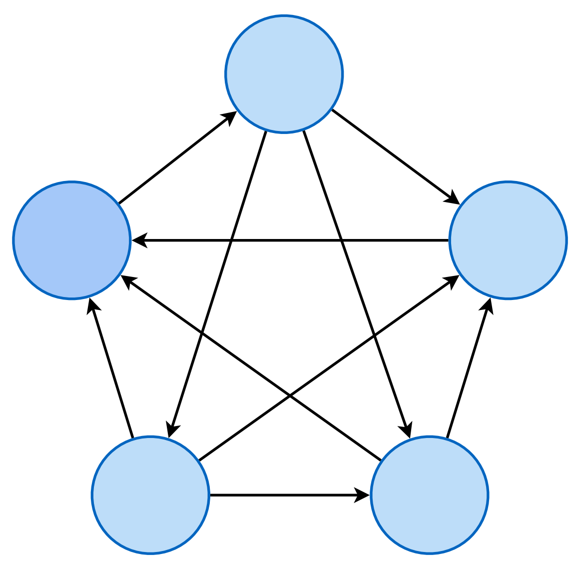
CloudComp + Inv

$\vdash$

$\{\mathsf{True}\}$

`r ← compute_factor(n);`

$\{\mathtt{r} = \mathsf{factor}(\mathtt{n})\}$

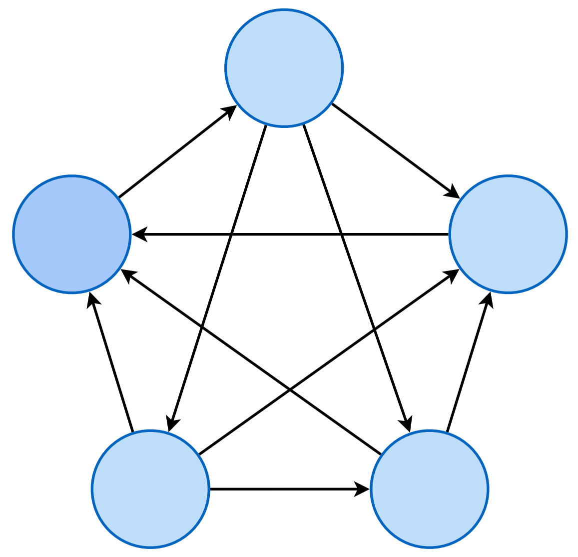


$\vdash$

{True}

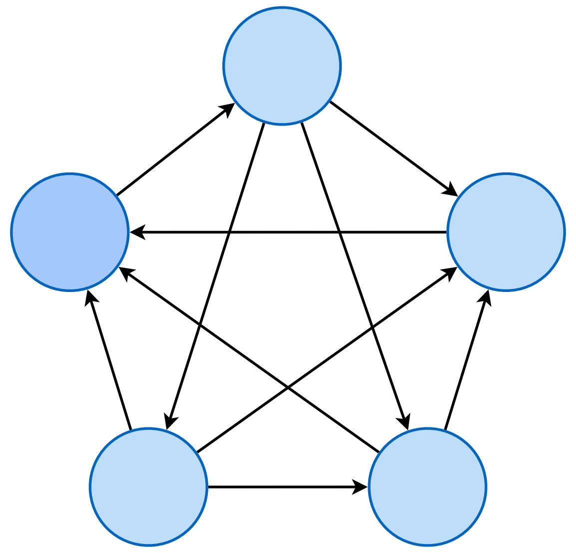
```
r ← compute_factor(n);
ps ← query_server(s)
```

{$\mathbf{r}$ = factor($\mathbf{n}$)}



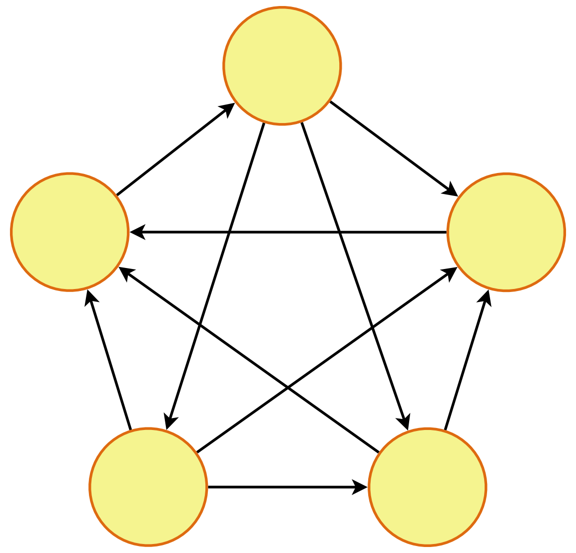
CloudComp + Inv

$$\vdash \quad \begin{array}{l} \{\mathrm{True}\} \\ \mathtt{r} \leftarrow \mathtt{compute\_factor(n);} \\ \mathtt{ps} \leftarrow \mathtt{query\_server(s)} \\ \{\mathbf{r} = \mathrm{factor}(\mathbf{n}) \wedge \mathtt{this} \notin \mathrm{ps}\} \end{array}$$



CloudComp + Inv
(CCI)

$\oplus$



QueryService
(QS)

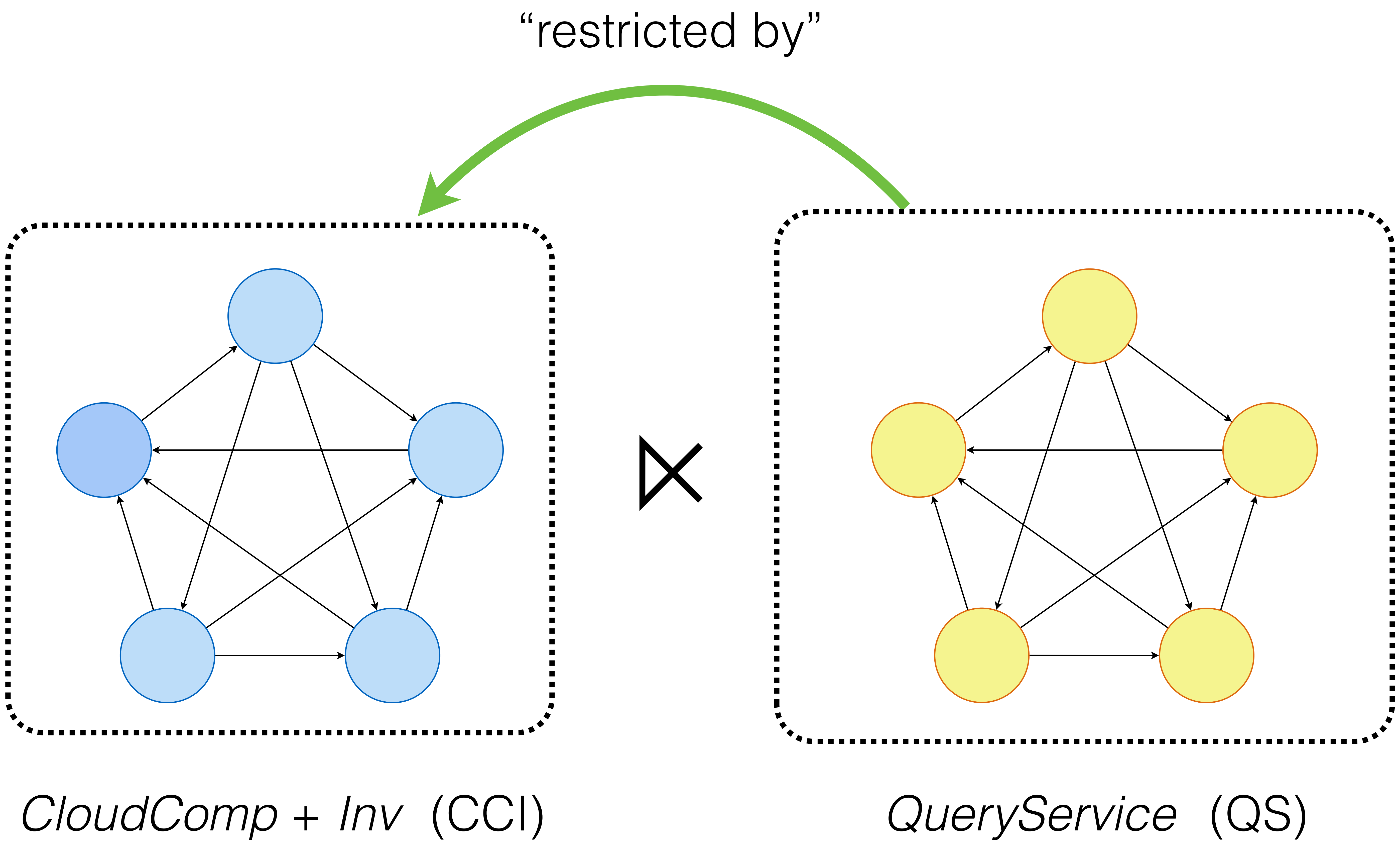
$\vdash$

$\{\text{True}\}$

```
r ← compute_factor(n);
ps ← query_server(s)
```

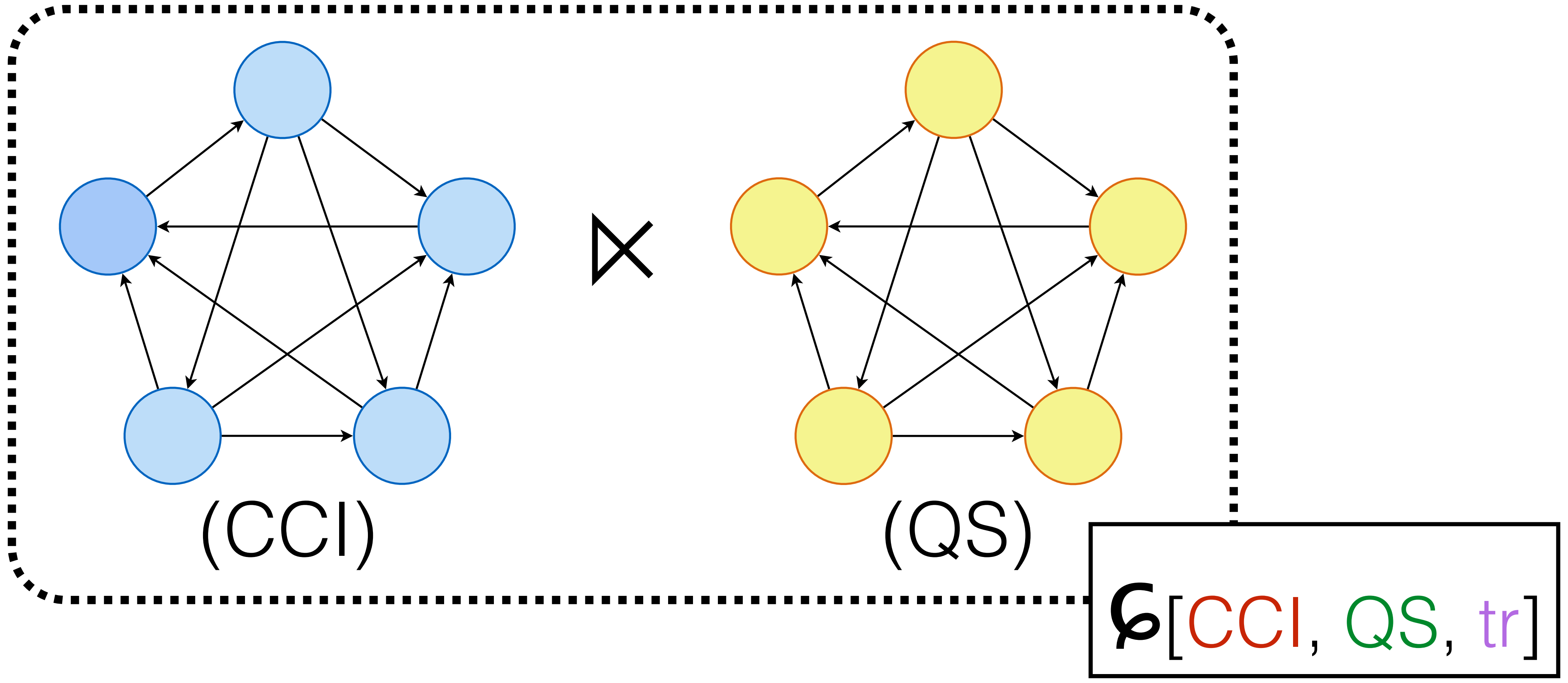
$\{\mathtt{r} = \text{factor}(\mathtt{n}) \land \mathtt{this} \notin \text{ps}\}$

Why adequate wrt. CCI?



“restricted by”
⋉
CloudComp + Inv (CCI)
QueryService (QS)

Logical Hooks

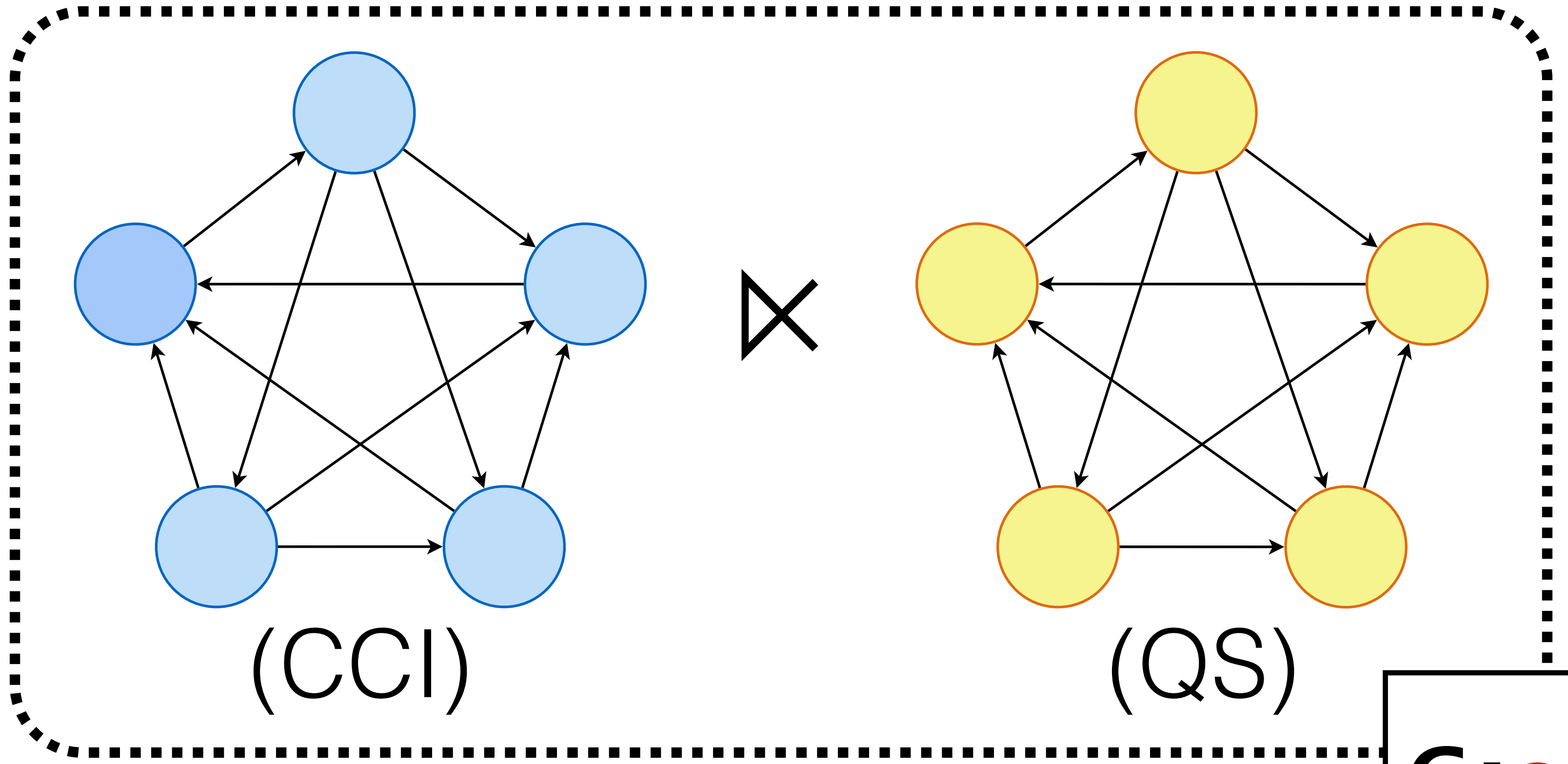


$$\wp_{[\mathrm{CCI},\ \mathrm{QS},\ \mathrm{tr}]}(\mathrm{m},\ \mathrm{s}_{\mathrm{cci}},\ \mathrm{s}_{\mathrm{qs}}) \triangleq$$

$$\mathrm{tr} \in \mathrm{QS}\ \wedge$$

$$\mathrm{tr}\ \textit{is send-response-to-enquiry}\ \wedge$$

$$\mathrm{m} = \mathrm{perms}(\mathrm{s}_{\mathrm{cci}})$$

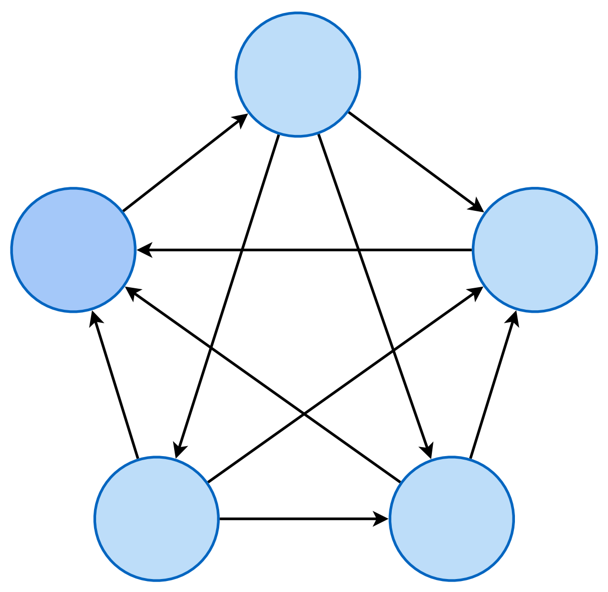


$\wp$[CCI, QS, tr]

$\vdash$

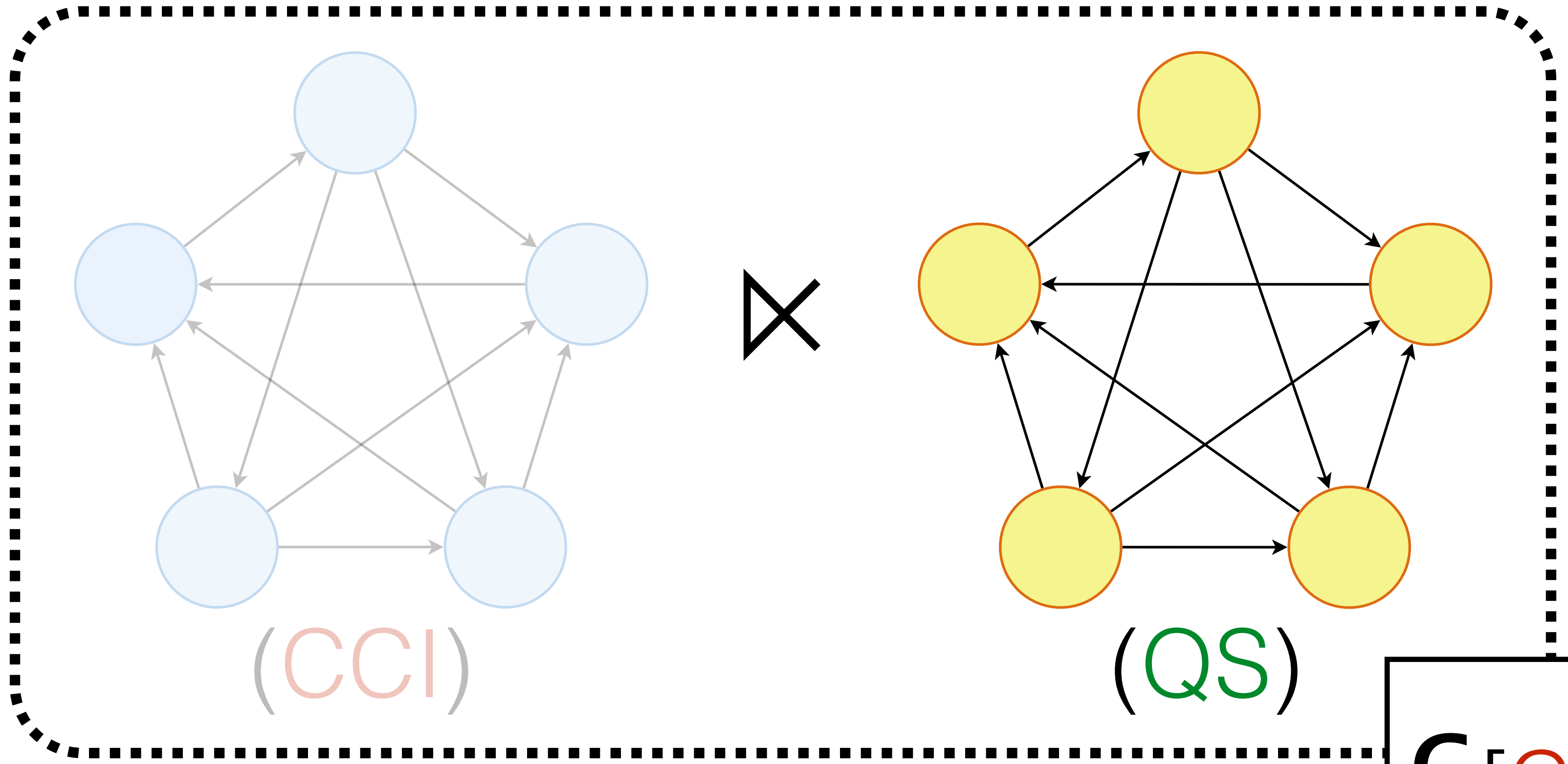
```
r ← compute_factor(n);
ps ← query_server(s)
```

$\{\mathbf{r} = \mathrm{factor}(\mathbf{n}) \wedge \mathtt{this} \notin \mathrm{ps}\}$



$$\{\text{True}\}$$
$$\vdash \texttt{r} \leftarrow \texttt{compute\_factor(n);}$$
$$\{\mathbf{r} = \text{factor}(\mathbf{n})\}$$

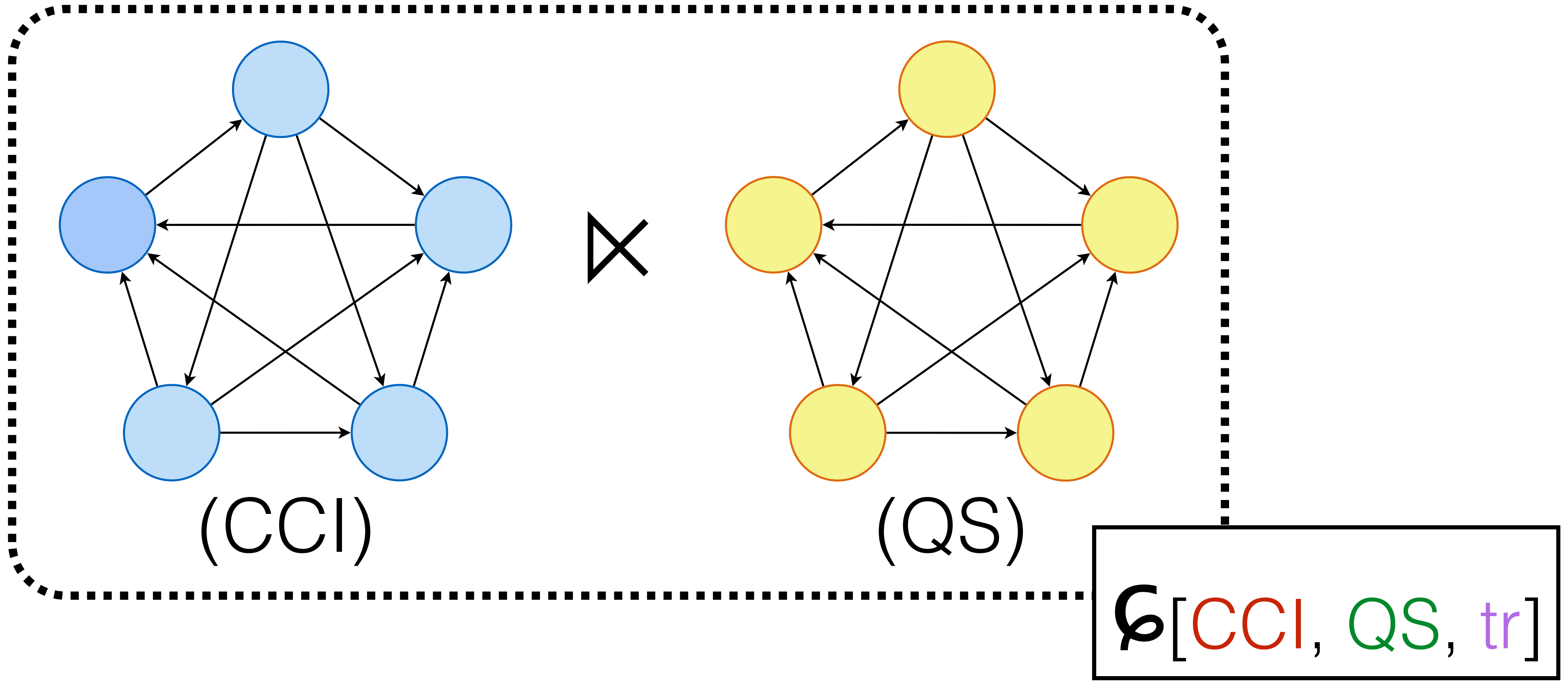
$$\wedge$$



$\mathcal{G}$[CCI, QS, tr]

$$\{\text{True}\}$$
$$\vdash \texttt{ps} \leftarrow \texttt{query\_server(s)}$$
$$\{\texttt{this} \notin \text{ps}\}$$

Hooks and Framing



- Hooks allow to *reuse* complex protocol invariants for server (dependable) components (e.g., CCI);

- *Hook Footprint* (*e.g.*, CCI) determines necessary server protocol that cannot be "framed out";

- Can be more *fine-grained*: consider *specific transitions*.

Composition in Distributed Systems

- Modular Program Verification

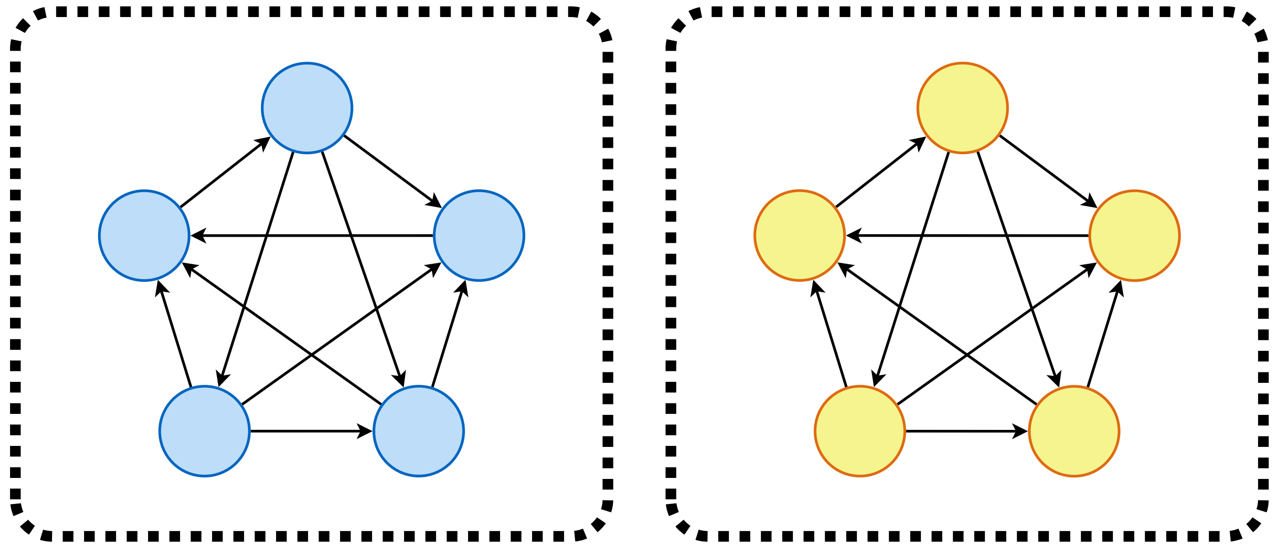
Protocol-aware logic +
Rule for *inductive invariants*

```
let f x = send x to h;
          r ← receive_from h
          return r
 in (f 42) + (f 239)
```

- Horizontal System Decomposition

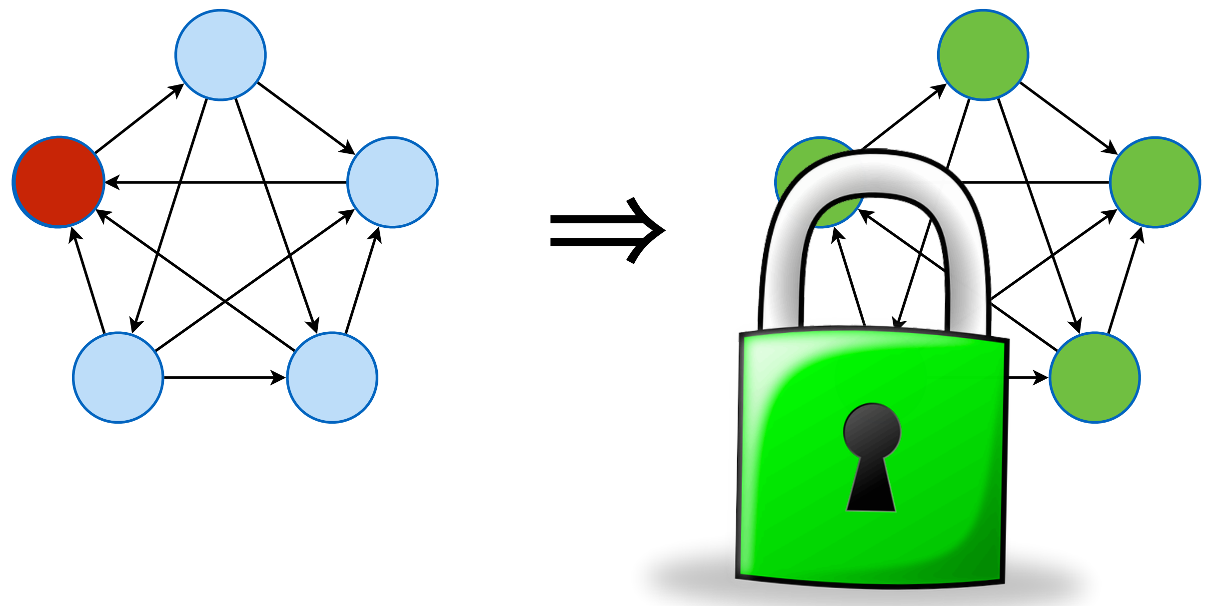
Framing wrt. a protocol

foo x ; bar y

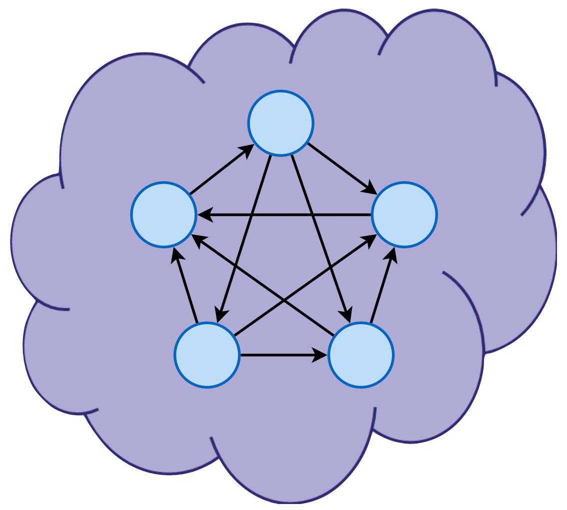


- Inter-Protocol Dependencies

Send-Hooks/Hook Footprint



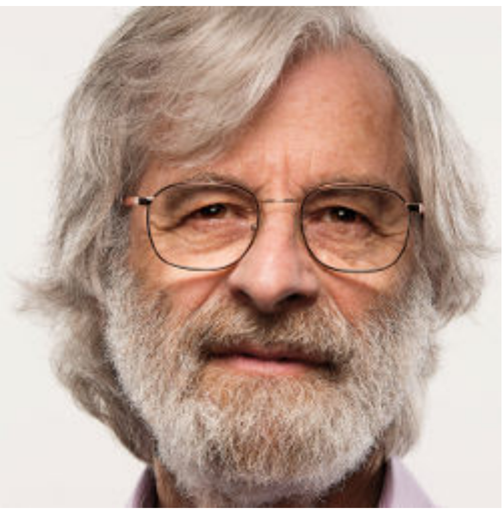
Disel:
Distributed Separation Logic



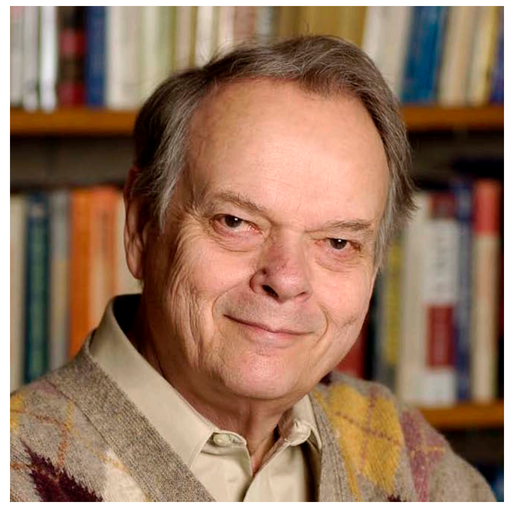
$$\vdash \{P\}\ c\ \{Q\}$$

https://github.com/DistributedComponents/disel

- Cloud Compute + Variations;
- Two-Phase Commit: Protocol, Invariants, Clients;
- Simple Blockchain Consensus protocol;
- Lease-based lock and distributed resource (WIP);
- Extraction and trusted shim implementation,

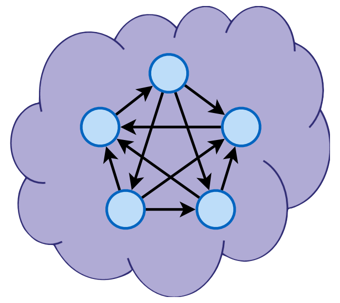


To Take Away

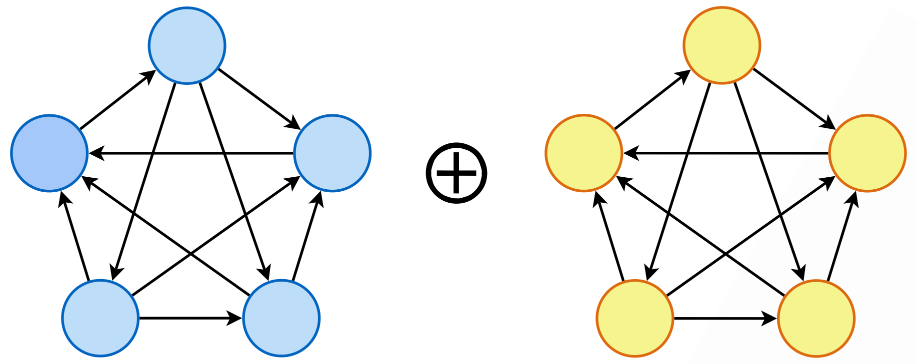


Compositional Reasoning about Distributed Systems

- *Separation* of Programs and Protocols: *Program Logics*

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

- *Separation* of Invariant Proofs: *Framing*



- *Separation* of Inter-Protocol Dependencies: *Hooks*

Plenty of aspects to address in the future:
node crashes, reconfiguration, byzantine faults, protocol updates, authentication, per-node concurrency, dynamic network topologies, integrating automation tools, (Ivy, TLA+, CVC4)…

Thanks!

Backup Slides

How is it different from (Multiparty) Session Types?

- Session types do not describe the state of nodes;
- No way to express global system invariants (*e.g.*, consensus);
- Limited support for horizontal system composition.

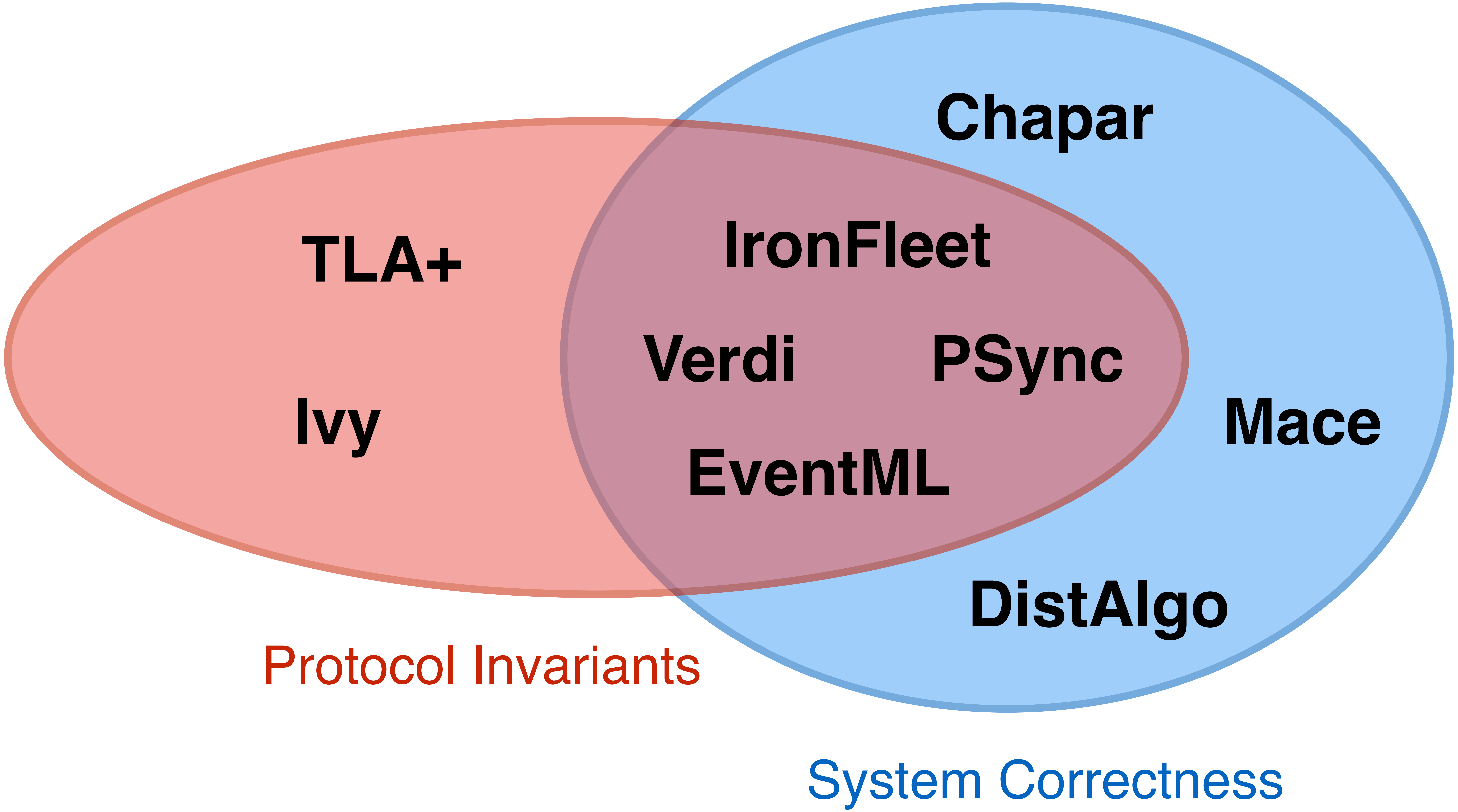
How is it different from proving program refinement?

- Our logic establishes a version of refinement
  by means of “programming with linearization points”;

- Protocol transitions (send/receive) — observable LPs.

- Information hiding by means of abstract predicates.

Verification Efforts



Verification Efforts

	Protocol-implementation modularity	Modular program verification	Horizontal protocol composition
IronFleet	**Yes**	**Sort of**	**No**
Verdi	**No**	**No**	**No**
PSync	**No**	**No**	**No**
EventML	**No**	**No**	**No**

Protocol Framing with Hooks

$$\frac{\begin{array}{c}\textsc{Frame} \\ \Gamma; W \overset{n}{\vdash} c : \{P\}\{Q\} \\ \mathsf{NotHooked}(W, H) \quad R \text{ is } C\text{-stable}\end{array}}{\Gamma; W \uplus \langle C, H\rangle \overset{n}{\vdash} c : \{P * R\}\{Q * R\}}$$

$$\textsc{Bind}\;\dfrac{\Gamma; W \overset{n}{\vdash} c_1 : \{P\}\{Q \wedge \mathsf{res} : \mathcal{T}\} \qquad \Gamma, x : \mathcal{T}; W \overset{n}{\vdash} [x/\mathsf{res}]c_2 : \{Q\}\{R\} \quad x \notin \mathsf{FV}(R)}{\Gamma; W \overset{n}{\vdash} x \leftarrow c_1; c_2 : \{P\}\{R\}}$$

$$\textsc{Letrec}\;\dfrac{\Gamma, x : \mathcal{T}, f : \langle W, \forall x : \mathcal{T}.\ \{P\}\{Q\}\rangle; W \overset{n}{\vdash} c : \{P\}\{Q\}}{\Gamma; W \overset{n}{\vdash} \mathbf{letrec}\ f(x : \mathcal{T}) \triangleq c : \forall x.\mathcal{T}.\ \{P\}\{Q\}}$$

$$\textsc{SendWrap}\;\dfrac{P, Q \text{ are } W\text{-stable} \quad W = \langle C, H\rangle \quad \tau_s \in C(\ell).T_s \qquad \mathsf{Sent}(\tau_s, \ell, n, m, to, H) \sqsubseteq (P, Q)}{\Gamma; W \overset{n}{\vdash} \mathsf{send}[\tau_s, \ell](m, to) : \{P\}\{Q\}}$$

$$\textsc{ReceiveWrap}\;\dfrac{P, Q \text{ are } W\text{-stable} \quad W = \langle C, H\rangle \qquad \mathsf{Received}(T, L, C) \sqsubseteq (P, Q)}{\Gamma; W \overset{n}{\vdash} \mathsf{recv}[T, L](m, to) : \{P\}\{Q\}}$$

$$\textsc{Read}\;\dfrac{P, Q \text{ are } W\text{-stable} \quad W = \langle C, H\rangle \qquad (\mathsf{this}\ s \wedge v \in \mathsf{dom}(s(\ell)(n)), \mathsf{this}\ s \wedge \mathsf{res} = s(\ell)(n)(v)) \sqsubseteq (P, Q)}{\Gamma; W \overset{n}{\vdash} \mathsf{read}_\ell(v) : \{P\}\{Q\}}$$

$$\textsc{Frame}\;\dfrac{\Gamma; W \overset{n}{\vdash} c : \{P\}\{Q\} \qquad \mathsf{NotHooked}(W, H) \quad R \text{ is } C\text{-stable}}{\Gamma; W \uplus \langle C, H\rangle \overset{n}{\vdash} c : \{P * R\}\{Q * R\}}$$

$$\textsc{WithInv}\;\dfrac{\Gamma; \langle \ell \mapsto \mathcal{P}_\ell \uplus W, H\rangle \overset{n}{\vdash} c : \{P\}\{Q\} \qquad I \text{ is inductive } \textit{wrt.}\ \mathcal{P}_\ell \qquad \mathcal{I} \triangleq \forall s, \mathsf{this}\ s \Rightarrow I(s)}{\Gamma; \langle \ell \mapsto \mathsf{WithInv}(\mathcal{P}_\ell, I) \uplus W, H\rangle \overset{n}{\vdash} c : \{P \wedge \mathcal{I}\}\{Q \wedge \mathcal{I}\}}$$

Network Semantics

$$\frac{\begin{array}{c} W = \langle C, H\rangle \qquad W \vDash s \qquad \ell \in \mathsf{dom}(C) \qquad \mathcal{P}_\ell = C(\ell) \qquad (MS, d) = s(\ell) \qquad \Big\{n,\ to\Big\} \subseteq \mathsf{dom}(d) \\ \tau_s \in \mathcal{P}_\ell.T_s \qquad \tau_s.pre(n, to, m, d) \qquad \mathsf{HooksOk}(W, \tau_s, \ell, s, n, m, to) \qquad MS' = MS \uplus \langle n, to, \circ, (\tau_s.tag, m)\rangle \end{array}}{s \ \leadsto^n_W \ s[\ell \mapsto (MS', d[n \mapsto \tau_s.step(to, m, d(n))])]} \ \textsc{Send}$$

$$\frac{\begin{array}{c} W = \langle C, H\rangle \quad W \vDash s \qquad \ell \in \mathsf{dom}(C) \qquad \mathcal{P}_\ell = C(\ell) \qquad (MS, d) = s(\ell) \qquad \tau_r \in \mathcal{P}_\ell.T_r \qquad MS = MS' \uplus \mathsf{m} \\ \mathsf{m} = \langle from, n, \circ, (\tau_r.tag, m)\rangle \qquad \{from, n\} \subseteq \mathsf{dom}(d) \quad \tau_r.pre(\mathsf{m}, d(n)) \qquad MS'' = MS' \uplus \langle from, n, \bullet, (\tau_r.tag, m)\rangle \end{array}}{s \ \leadsto^n_W \ s[\ell \mapsto (MS'', d[n \mapsto \tau_r.step(\mathsf{m}, d(n))])]} \ \textsc{Recv}$$

Component	Defs/Specs	Impl	Proofs	Build
Calculator (§2)				
protocol (§2.1) $\textsc{Inv}_1$ (§2.3) $\textsc{Inv}_2$ (§2.4)	239	-	243	4.8
simple_server (§2.3) batch_server (§2.4) memo_server (§2.4)	192	43	153	8.6
compute (§2.4) deleg_server (§2.4)	120 75	24 7	99 49	4.8 2.4
Two-Phase Commit (§4.1–§4.3)				
protocol (§4.1)	465	-	231	3.9
coordinator (§4.2)	236	35	440	20
participant (§4.2)	163	24	198	11
`TPCInv` (§4.3)	997	-	2113	36
Query/TPC (§4.4)				
protocol	169	-	115	2.1
querying procedures	326	18	707	22
`run_and_query`	76	5	89	2.6