Overview of the Reowolf Project

Christopher Esterhuyse

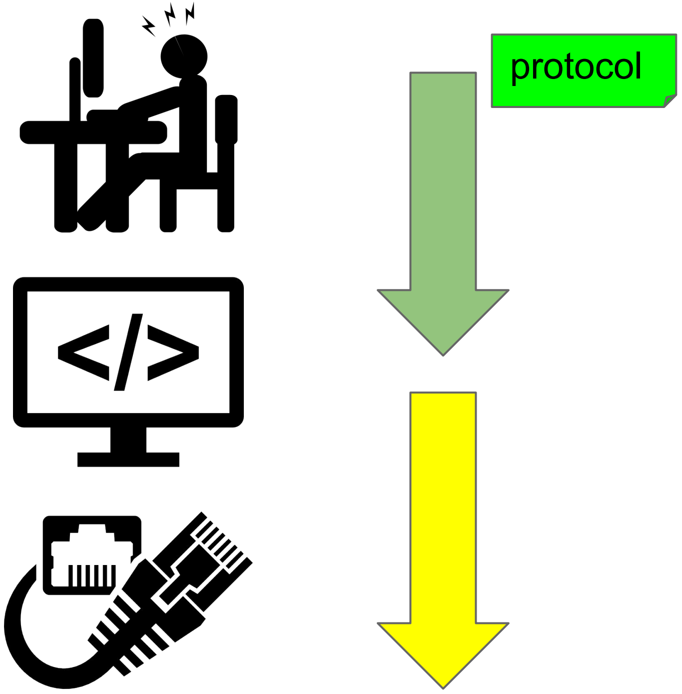
Contents

Part 1/4:
Context

Context: Problem

State of socket programming

- BSD-style sockets are very limited
  - 2-party communication
  - Limited configurability
- High-level logic → low-level implementation
  - Error prone for humans
  - Over-specifies original requirements
  - Original intention is lost
- Middleware is ignorant of the protocol
  - Uninformed resource optimization
  - Cannot help preserve requirements

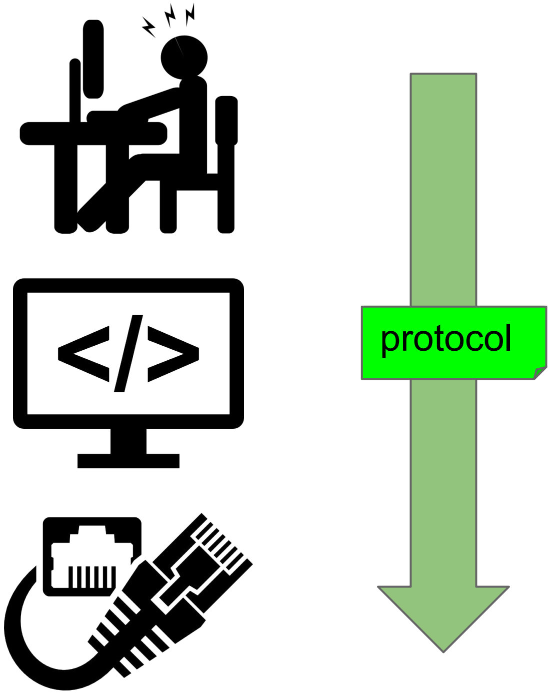


Context: Approach

Use explicit **protocols** as the vehicle for the user's requirements, preserved all the way down to the infrastructure

Project deliverables:

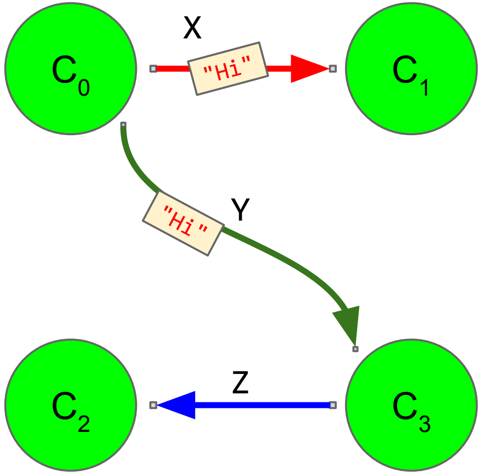
1. Protocol Description Language (**'PDL'**)
2. Implementation of **connectors**, configurable with protocols expressed in PDL



Part 2/4:
Usage

Usage: Connectors & Sessions

A **session** is a particular run of a system of communicating **components**, communicating via the exchange of **messages** over time.



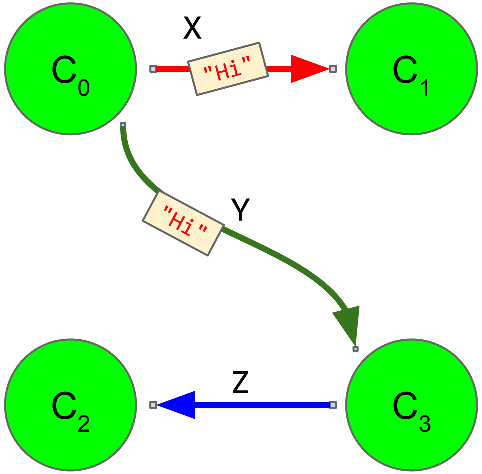
Usage: Connectors & Sessions

A **session** is a particular run of a system of communicating **components**, communicating via the exchange of **messages** over time.

We discretize time into a sequence of **interactions**.

	X	Y	Z
0	"Hi"	"Hi"	*
1	*	*	*

at round 0:



Usage: Connectors & Sessions

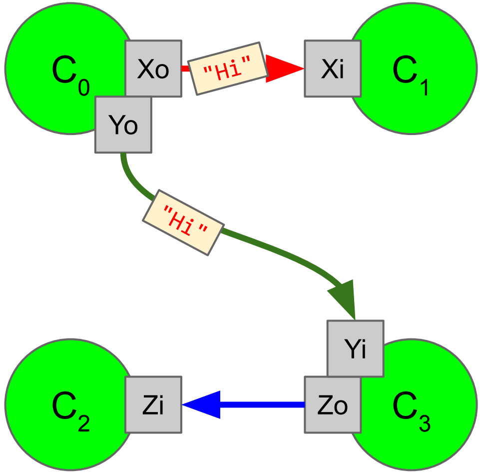
A **session** is a particular run of a system of communicating **components**, communicating via the exchange of **messages** over time.

We discretize time into a sequence of **interactions**.

	Xo	Xi	Yo	Yi	Zo	Zi
0	"Hi"	"Hi"	"Hi"	"Hi"	*	*
1	*	*	*	*	*	*

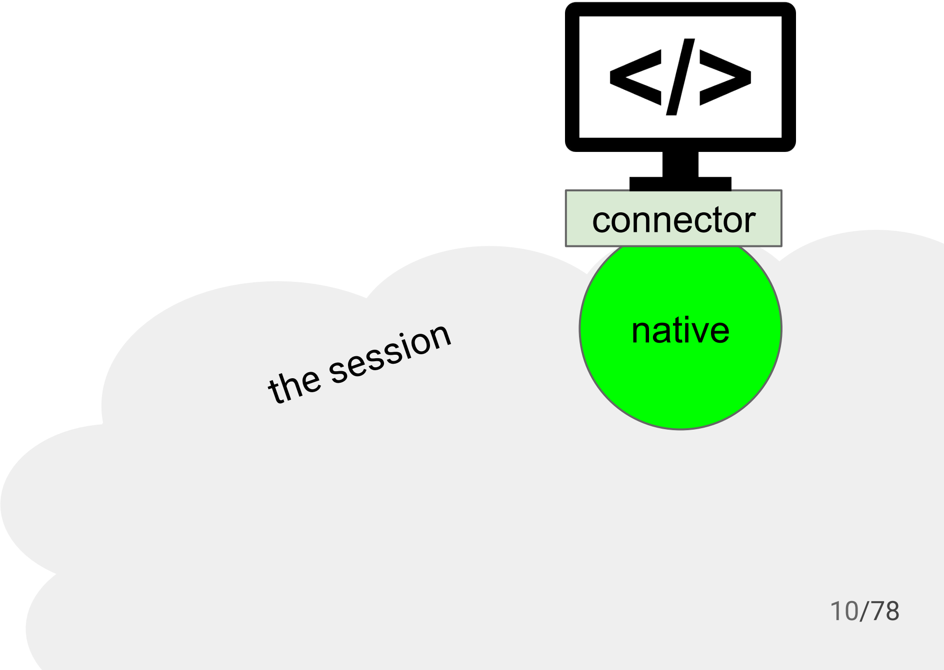
Components act on **ports** (~channel ends), so we often reason at this granularity. Components only access their own ports.

at round 0:



Usage: Connectors & Sessions

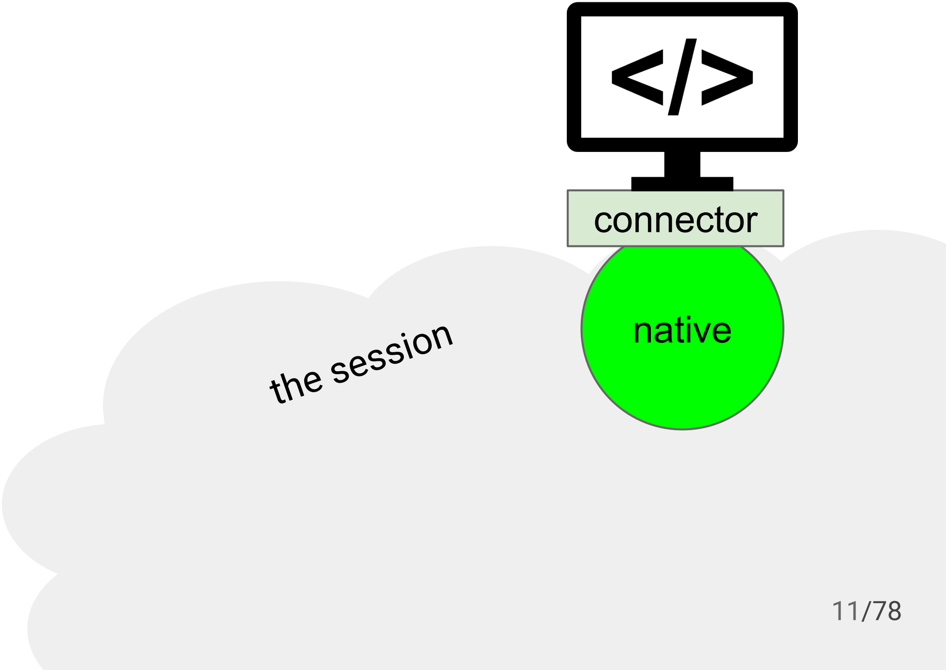
Connectors allow an application to participate in a session, adopting the role of a **native** (component).



Usage: Connectors & Sessions

Connectors allow an application to participate in a session, adopting the role of a **native** (component).

The session starts after a **setup** phase, in which the application refines the session configuration around their native component.

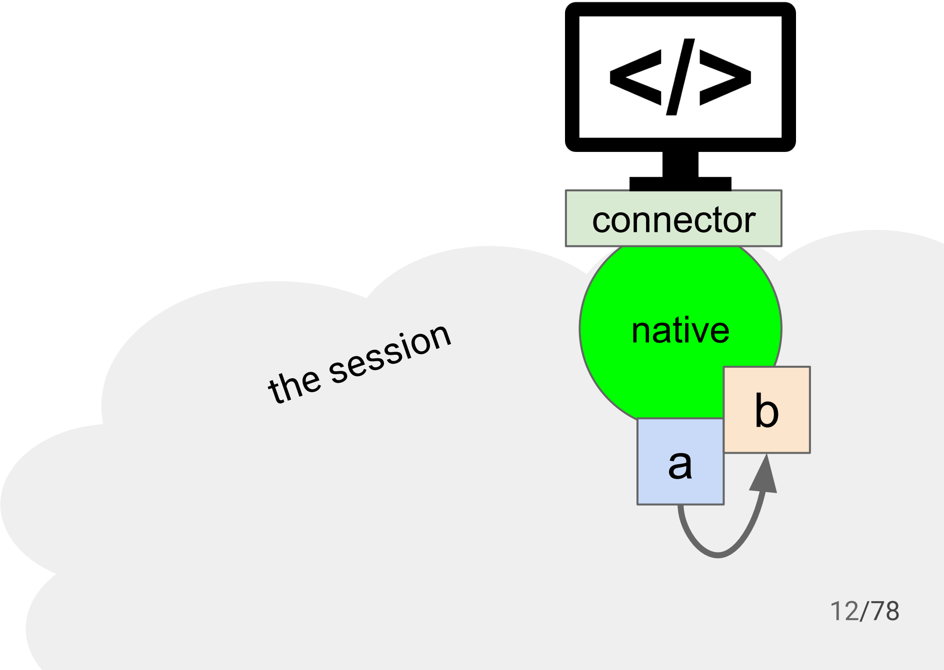


Usage: Setup

Connectors allow an application to participate in a session, adopting the role of a **native** (component).

The session starts after a **setup** phase, in which the application refines the session configuration around their native component. They can:

1. Create channels, keeping both ports

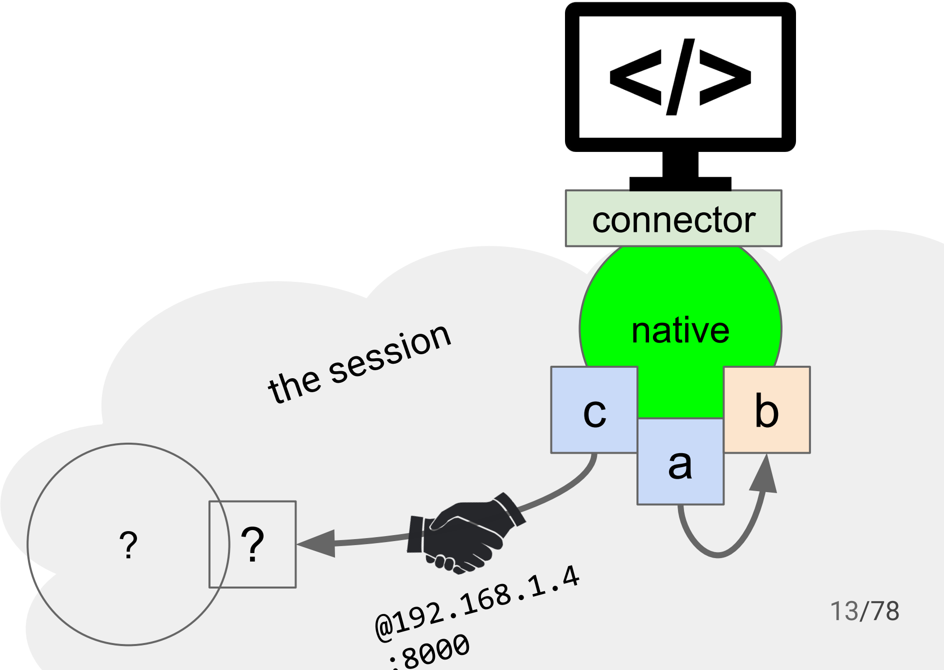


Usage: Setup

Connectors allow an application to participate in a session, adopting the role of a **native** (component).

The session starts after a **setup** phase, in which the application refines the session configuration around their native component. They can:

1. Create channels, keeping both ports
2. Cooperate with a peer to create a channel



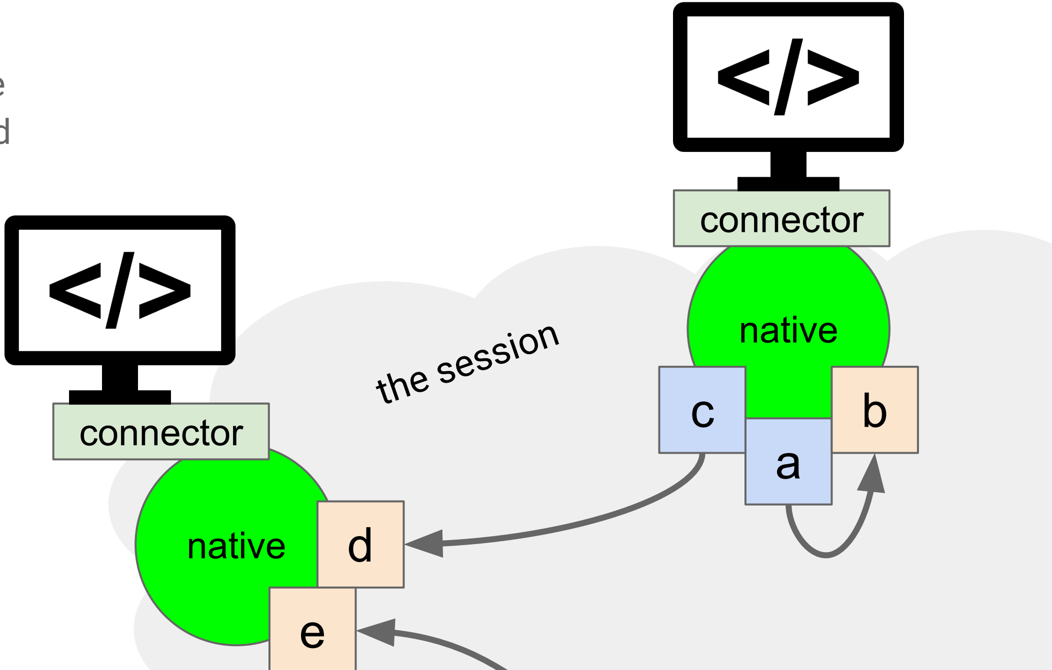
Usage: Setup

Connectors allow an application to participate in a session, adopting the role of a **native** (component).

The session starts after a **setup** phase, in which the application refines the session configuration around their native component. They can:

1. Create channels, keeping both ports
2. Cooperate with a peer to create a channel

All connectors transition setup→communication together, when they complete **`connect()`**.

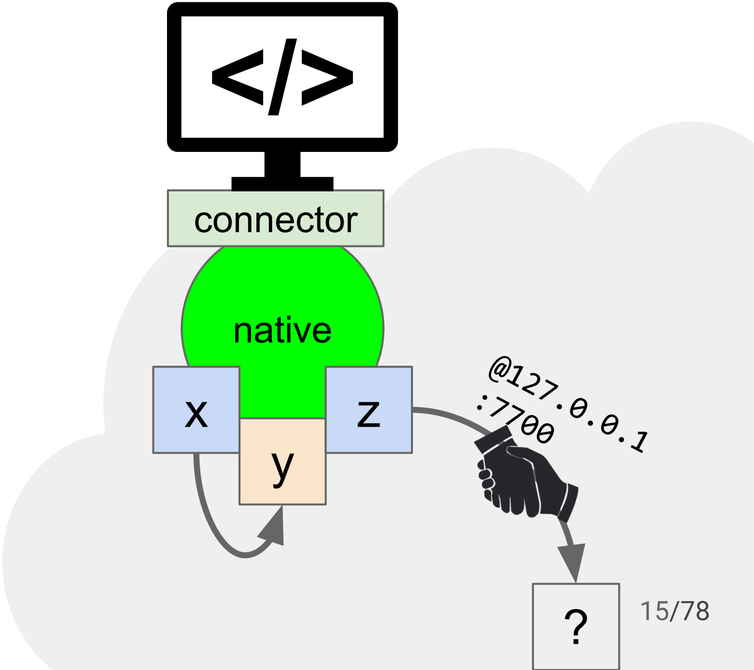


Usage: Setup

Example C code of setup phase:

Ignore this `config` for now

```
Connector * c = connector_new(config);

PortId x, y, z;
connector_add_port_pair(c, &x, &y);
connector_add_net_port(c, &z,
  (FfiSocketAddr) {{127, 0, 0, 1}, 7700},
  Polarity_Putter,
  EndpointPolarity_Active);

connector_connect(c, -1);
```



Usage: Communication

Communication proceeds in **rounds** (~interactions), during which every port may send or receive up to 1 message. Components may work on local data `between' rounds.

The C API renders this as a *builder pattern*, where the application **synchronizes** local data with that of the session in rounds. In steps:

1. Prepare for the next synchronization
2. Synchronize message data
3. Reflect on the result

```
connector_put_bytes(c, x, "Hi", 2);
connector_get(c, y);
connector_put_bytes(c, z, "Hey", 3);

connector_sync(c, -1);

size_t len;
const unsigned char * msg =
  connector_gotten_bytes(c, x, &len);
```

Usage: Communication

Components can express **nondeterministic** choice, to be decided arbitrarily at runtime.

For native components: group messages into indexed '**batches**'; exactly one batch will succeed.

Batch 0

Batch 1

```
connector_put_bytes(c, x, "Hey", 3);
connector_get(c, y);

connector_next_batch(c);

connector_put_bytes(c, x, "Hi", 2);

int code = connector_sync(c, -1);

switch(code) {
  case  0: /*              */ break;
  case  1: /*              */ break;
  default: /* (error case) */ break;
}
```

Usage: Communication

Components can express **nondeterministic** choice, to be decided arbitrarily at runtime.

For native components: group messages into indexed ‘**batches**’; exactly one batch will succeed.

Why? Component can be flexible to other components’ behavior without knowing it.

Batch 0

```
connector_put_bytes(c, x, "Hey", 3);
connector_get(c, y);

connector_next_batch(c);
```

Batch 1

```
connector_put_bytes(c, x, "Hi", 2);

int code = connector_sync(c, -1);

switch(code) {
  case  0: /*              */ break;
  case  1: /*              */ break;
  default: /* (error case) */ break;
}
```

Usage: Communication

Example 1-round session

```
Connector * c = connector_new(config);
PortId p;
connector_add_net_port(c, &p, addr,
  Polarity_Putter, EndpointPolarity_Active);
connector_connect(c, -1);

connector_put_bytes(c, p, "Hi", 2);
connector_next_batch(c);
int err = connector_sync(c, 1000);
if(code == 1) {
  // my message was sent!
}
```

(this native *offers* the message: `"Hi"`)

```
Connector * c = connector_new(config);
PortId g;
connector_add_net_port(c, &g, addr,
  Polarity_Getter, EndpointPolarity_Passive);
connector_connect(c, -1);


connector_get(c, g);
connector_sync(c, 1000);
size_t len;
const char msg =
  connector_gotten_bytes(c, g, &len);
printf("%.*s\n", (int) len, msg);
```

(this native *demands* some message)

Usage: Protocols

Protocol Description Language (‘**PDL**’) defines **protocol components**, and aims to feel familiar to C programmers.

```
[redacted] foo(in a, in b, out c) {
  int counter = 0;
}
```

Usage: Protocols

Protocol Description Language (‘**PDL**’) defines **protocol components**, and aims to feel familiar to C programmers.

Primitive components can participate in rounds, putting or getting messages through ports.

```
primitive foo(in a, in b, out c) {
  int counter = 0;
  synchronous {
    msg ma = get(a);
  }
  synchronous {
    msg mb = get(b);
  }
}
```

Usage: Protocols

Protocol Description Language (‘**PDL**’) defines **protocol components**, and aims to feel familiar to C programmers.

Primitive components can participate in rounds, putting or getting messages through ports.

Unlike natives, protocol components can introduce **causal dependencies** between actions.

```
primitive foo(in a, in b, out c) {
  int counter = 0;
  synchronous {
    msg ma = get(a);
  }
  synchronous {
    msg mb = get(b);
    put(c, mb);
  }
}
```

Usage: Protocols

Protocol Description Language (‘**PDL**’) defines **protocol components**, and aims to feel familiar to C programmers.

Primitive components can participate in rounds, putting or getting messages through ports.

Unlike natives, protocol components can introduce **causal dependencies** between actions.

They can express nondeterminism by accessing values decided at runtime.

```
primitive foo(in a, in b, out c) {
  int counter = 0;
  synchronous {
    msg ma = get(a);
  }
  synchronous {
    if(fires(b) && fires(c)) {
      msg mb = get(b);
      put(c, mb);
    }
  }
}
```

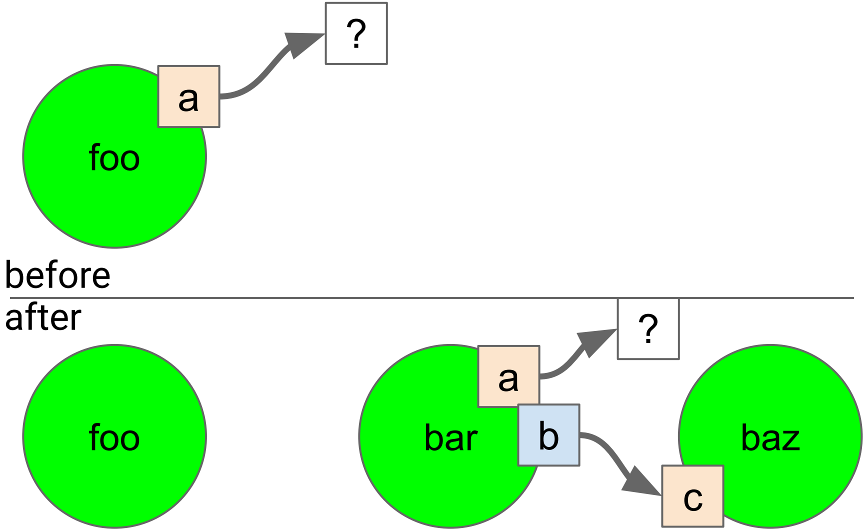
Usage: Protocols

Composite components can create new port pairs, but cannot communicate.

```
composite foo(in a) {
  channel b -> c;
}
```

Usage: Protocols

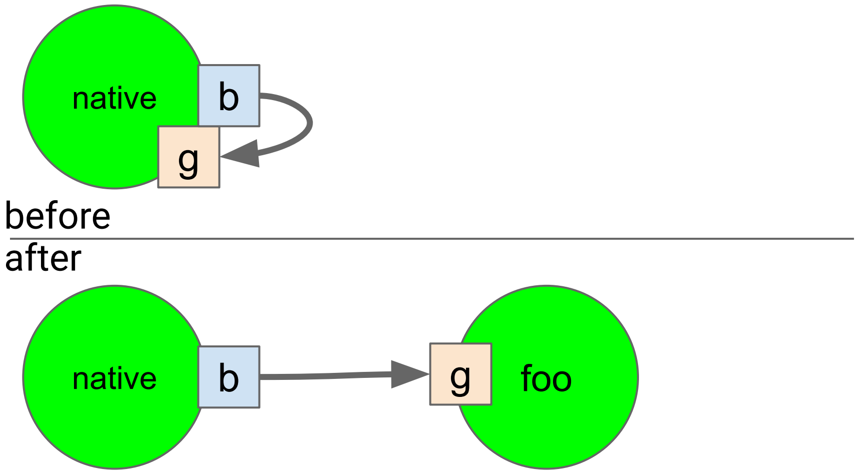
Composite components can create new port pairs, but cannot communicate. They can also create new components, and pass them their ports.




```
composite foo(in a) {
  channel b -> c;
  new bar(a, b);
  new baz(c);
}
primitive bar(in a, out b) {
  /* omitted */
}
composite baz(in c) {
  /* omitted */
}
```

Usage: Connectors + Protocols

Like composites, natives can create new protocol components, effectively delegating work to the connector itself.

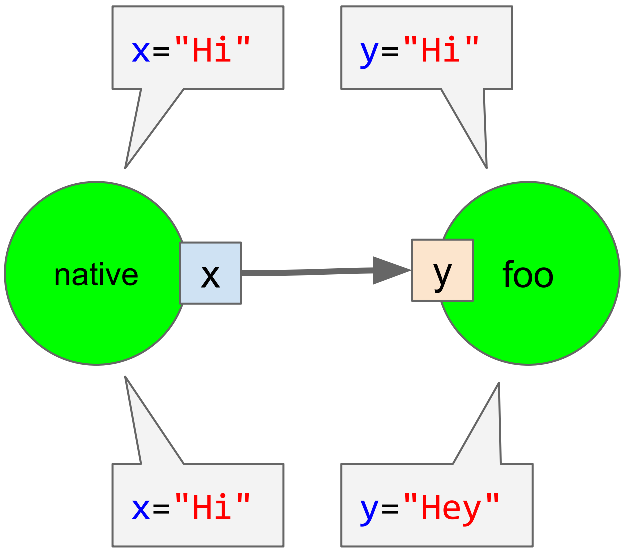



```
unsigned char pdl = "primitive foo(in g){}";
Arc_ProtocolDescription config =
  protocol_description_parse(pdl, sizeof(pdl)-1);
Connector * c = connector_new(config);

PortId p, g;
connector_add_port_pair(c, &p, &g);
connector_add_component(c, "foo", 3, &g, 1);

connector_connect(c, -1);
```

Usage: Session behavior

Connectors realize an **interaction** per round, where:

1. Components consense on the interaction
2. No component's constraints are violated



Usage: Session behavior

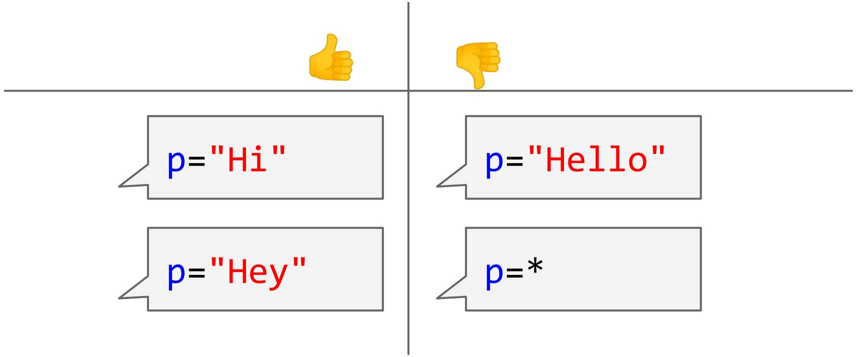
Connectors realize an **interaction** per round, where:

1. Components consense on the interaction
2. No component's constraints are violated

```
connector_put_bytes(c, p, "Hi", 2);
connector_next_batch(c);

connector_put_bytes(c, p, "Hey", 3);
connector_sync(c, -1);
```

For a **Native** component:

Messages exchanged through ports must match those expressed in *one* batch.



Usage: Session behavior

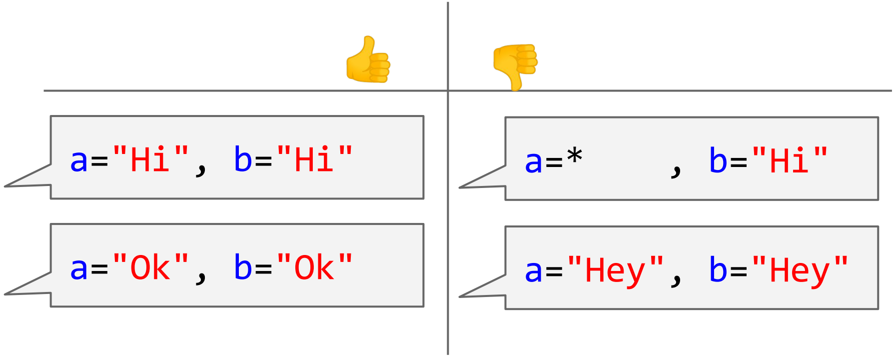
Connectors realize an **interaction** per round, where:

1. Components consense on the interaction
2. No component's constraints are violated

```
primitive foo(in a, out b) {
  synchronous {
    if(fires(a)) {
      msg m = get(a);
      put(b, m);
      assert(m.length == 2);
} } }
```

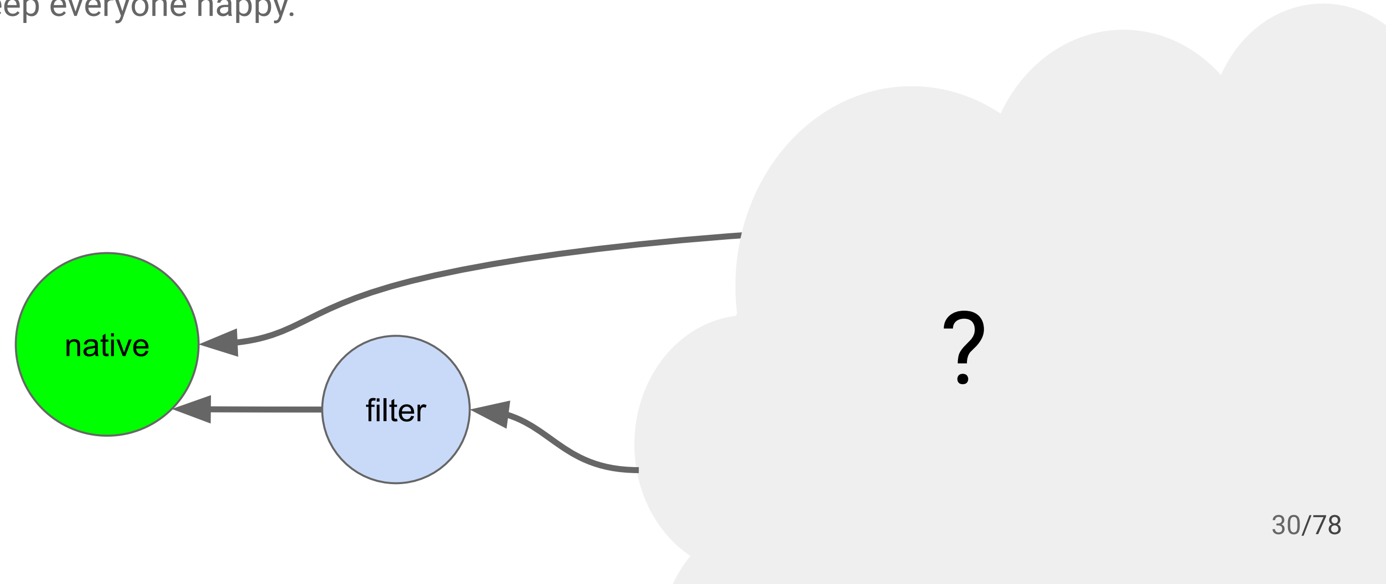
For a **Protocol** component:

The component's updated state is explained by a path through the synchronous block without errors



Usage: Session behavior

The idea:
Applications can express their requirements to the session, and then focus only on their local behavior, relying on the connector to keep everyone happy.



Part 3/4:
Internals

Part $3_a/4$: Internals: Interactions

Interactions: Constraint Satisfaction

The session is realized by the (distributed) **connector runtime** , comprised of the session's connectors.

Each round, the runtime solves a **distributed constraint satisfaction** problem

Interactions: Constraint Satisfaction

The session is realized by the (distributed) **connector runtime** , comprised of the session's connectors.

Each round, the runtime solves a **distributed constraint satisfaction** problem, where:

- The solution is an **interaction** & state update

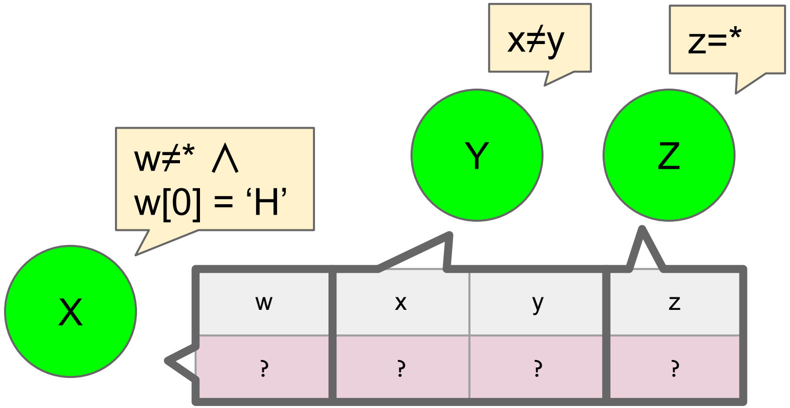
w	x	y	z
?	?	?	?

Interactions: Constraint Satisfaction

The session is realized by the (distributed) **connector runtime** , comprised of the session's connectors.

Each round, the runtime solves a **distributed constraint satisfaction** problem, where:

- The solution is an **interaction** & state update
- The **constraints** are given by components

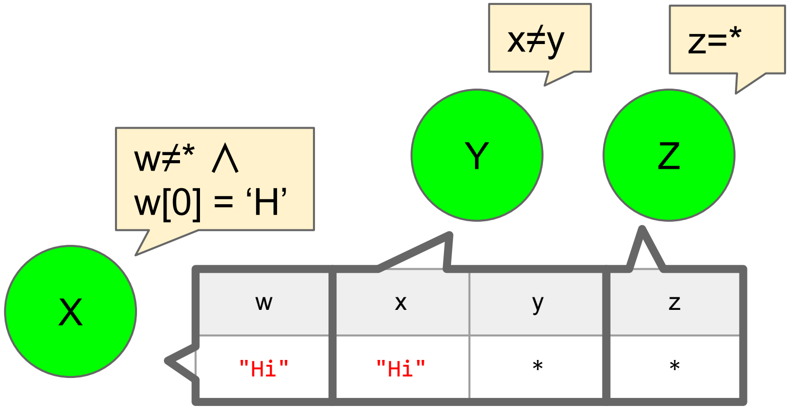


Interactions: Constraint Satisfaction

The session is realized by the (distributed) **connector runtime** , comprised of the session's connectors.

Each round, the runtime solves a **distributed constraint satisfaction** problem, where:

- The solution is an **interaction** & state update
- The **constraints** are given by components



Interactions: Constraint Satisfaction

Naïve constraint satisfaction:
Simply enumerate and check candidate solutions

w	x	y	z
*	*	*	*

w	x	y	z
*	*	*	00

w	x	y	z
*	*	00	00

•••

w	x	y	z
"Hi"	"Hi"	*	*

Interactions: Constraint Satisfaction

Naïve constraint satisfaction:
Simply enumerate and check candidate solutions

TODO: How do we “check” candidates?

w	x	y	z
*	*	*	*

w	x	y	z
*	*	*	00

w	x	y	z
*	*	00	00

• • •

w	x	y	z
"Hi"	"Hi"	*	*

Interactions: Candidate checking

A candidate is a solution IFF it “satisfies” every component.

Interactions: Candidate checking

A candidate is a solution IFF it “satisfies” every component.

- <u>Native components</u>:
  Port operations match one batch.

```
connector_put_bytes(c, w, "Hi", 2);
connector_sync(c, -1);
```

w	x	y	z
"Hi"	"Hi"	*	*

Interactions: Candidate checking

A candidate is a solution IFF it "satisfies" every component.

- <u>Native components</u>:
  Port operations match one batch.

- <u>Protocol components</u>:
  Interaction 'explains' a path through the synchronous block, and the updated state.

```
primitive foo(in x, out y, in z) {
  synchronous {
    get(x);
  }
}
```

```
connector_put_bytes(c, w, "Hi", 2);
connector_sync(c, -1);
```

w	x	y	z
"Hi"	"Hi"	*	*

Interactions: Candidate checking

A candidate is a solution IFF it “satisfies” every component.

- Native components:
  Port operations match one batch.

- Protocol components:
  Interaction ‘explains’ a path through the synchronous block, and the updated state.

TODO: How do we distribute this task?

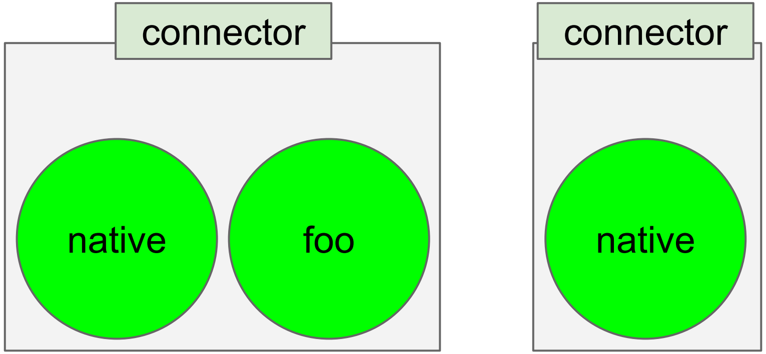
```
primitive foo(in x, out y, in z) {
  synchronous {
    get(x);
  }
}
```

```
connector_put_bytes(c, w, "Hi", 2);
connector_sync(c, -1);
```

w	x	y	z
"Hi"	"Hi"	*	*

Interactions: Solution Tree

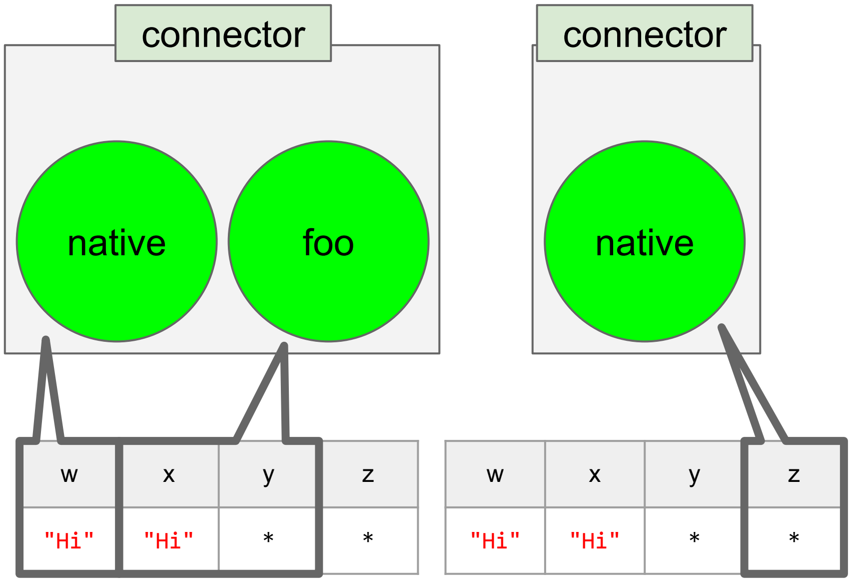
We partition components over the connectors;
we say connectors **manage** their components.



Interactions: Solution Tree

We partition components over the connectors;
we say connectors **manage** their components.

Component constraints are checked by its manager.

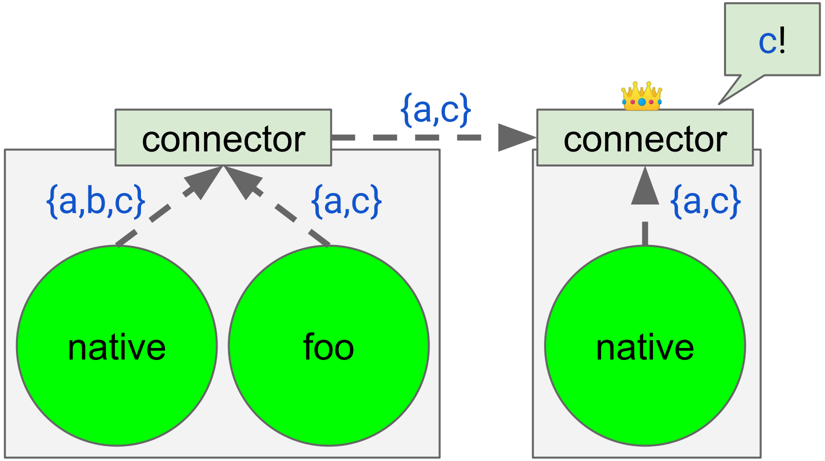


Interactions: Solution Tree

We partition components over the connectors;
we say connectors **manage** their components.

Component constraints are checked by its manager.

Candidates filter down the **solution tree**, whose root **decides**, and announces the solution.



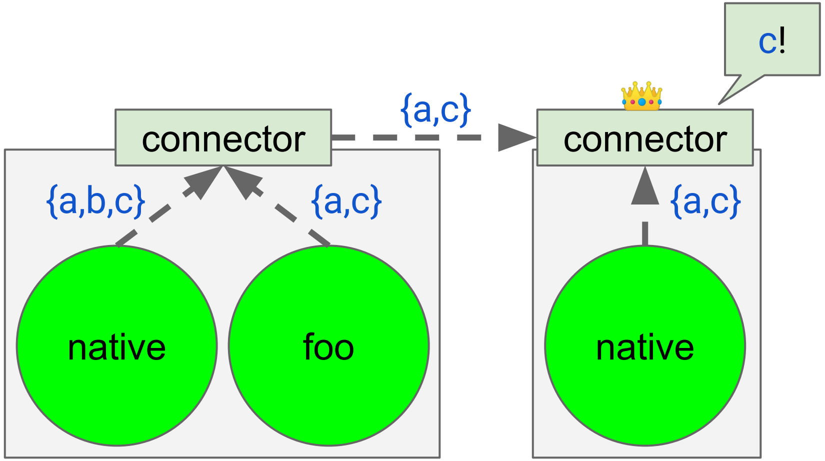
Interactions: Solution Tree

We partition components over the connectors;
we say connectors **manage** their components.

Component constraints are checked by its manager.

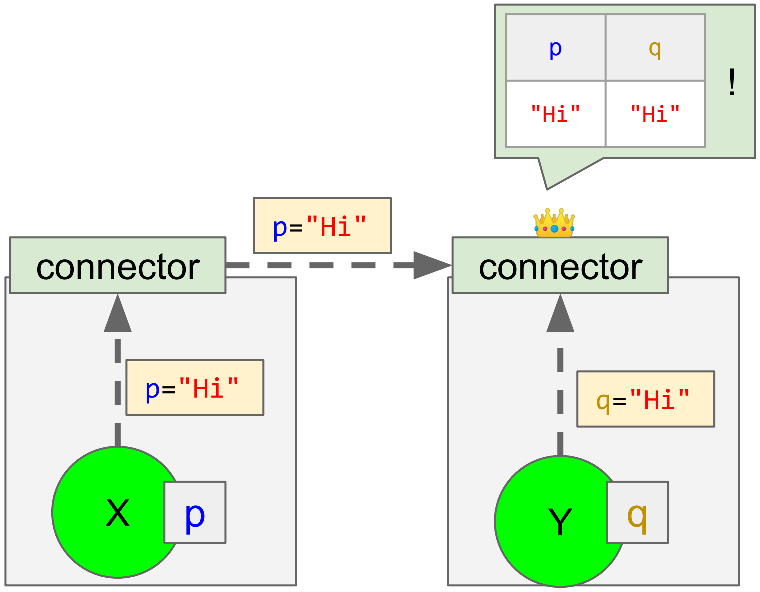
Candidates filter down the **solution tree**, whose root **decides**, and announces the solution.

TODO: How do we reduce candidate size & number?



Interactions: Candidate Predicates

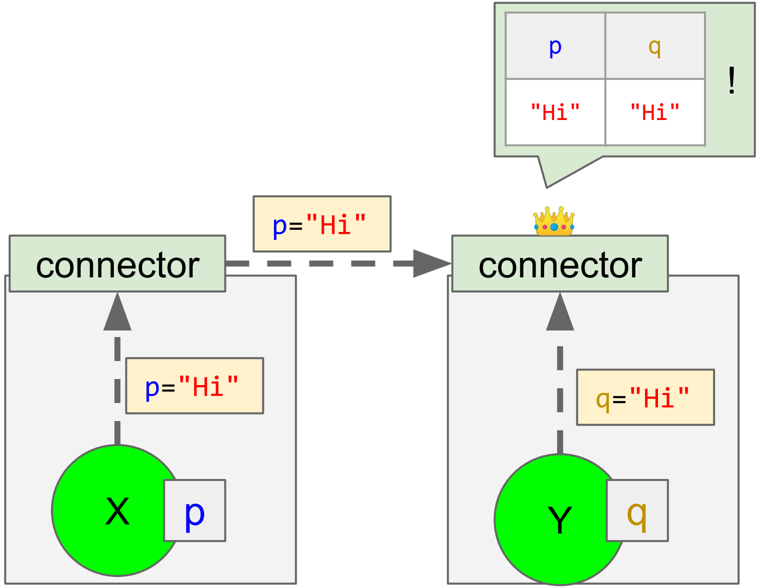
We introduce **candidate predicates**, a structure that tersely encodes a set of candidate solutions. Increasingly specific predicates filter to the root.



Interactions: Candidate Predicates

We introduce **candidate predicates**, a structure that tersely encodes a set of candidate solutions. Increasingly specific predicates filter to the root.

Intuition:
We exploit the fact that not all components constrain the values of a given port.

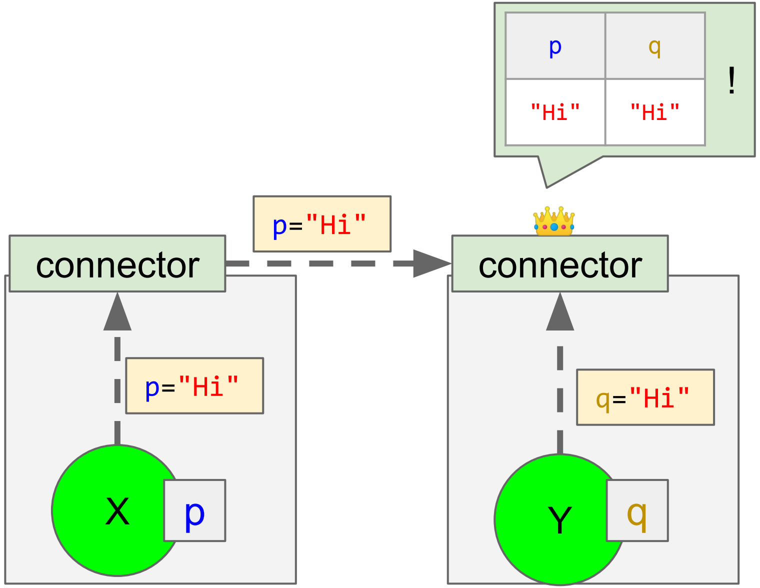


Interactions: Candidate Predicates

We introduce **candidate predicates**, a structure that tersely encodes a set of candidate solutions. Increasingly specific predicates filter to the root.

Intuition:
We exploit the fact that not all components constrain the values of a given port.

TODO: How to explore a component's candidates?



Interactions: Speculation

Rather than enumerating + checking candidates, we use available information to do these together.

Interactions: Speculation

Rather than enumerating + checking candidates, we use available information to do these together.

- Native components
  Connectors are explicitly told what options they allow; one predicate per *batch*.

```
connector_put_bytes(c, p, "Hi", 2);
connector_get(c, g);
connector_next_batch(c);

connector_put_bytes(c, p, "Hey", 3);
connector_sync(c, -1);
```

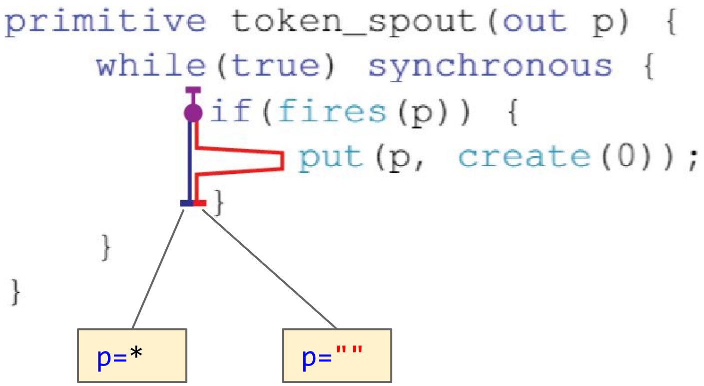
```
{ p="Hi"   p="Hey" }
  g≠*      g=*
```

Interactions: Speculation

Rather than enumerating + checking candidates, we use available information to do these together.

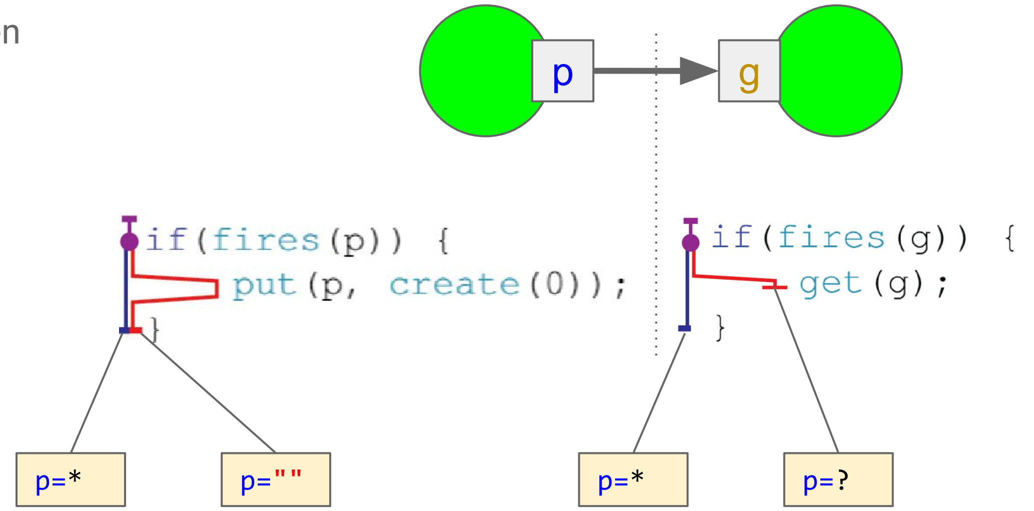
- <u>Native components</u>
  Connectors are explicitly told what options they allow; one predicate per *batch*.

- <u>Protocol components</u>
  Connectors **speculatively execute** protocol components, unfolding the *paths* through their synchronous blocks without error.



Interactions: Speculation

Rather predicates encoding relationships between message contents (gets very complex!) causally
Components *cooperate* during speculation

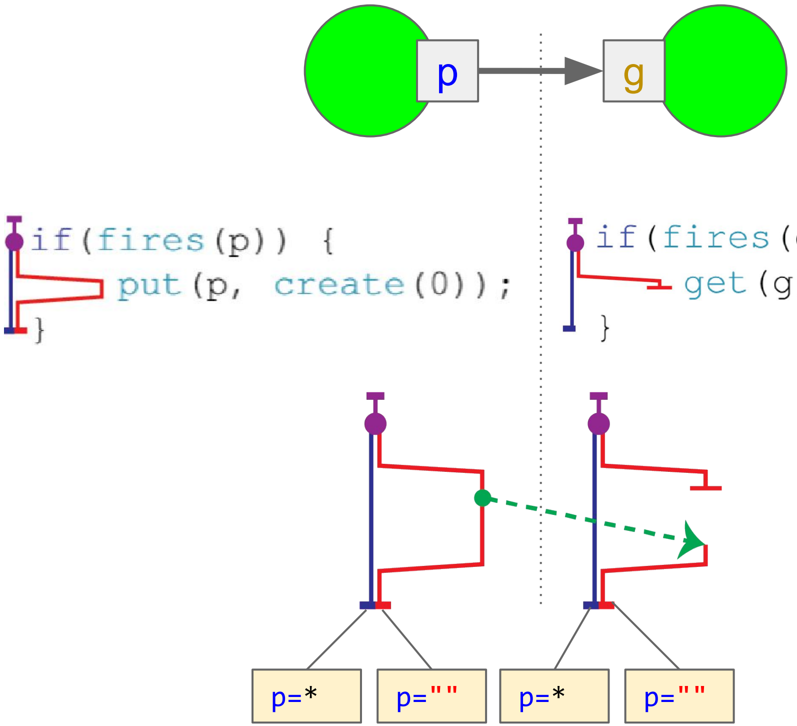


Interactions: Speculation

Rather predicates encoding relationships between message contents (gets very complex!) causally
Components *cooperate* during speculation:
Putters inform getters of **speculative messages**.

The idea:
The cost of speculation scales with satisfactory *paths* through components, and not with satisfactory *values* of messages.

Bonus:
Speculation is *lazy*, delayed until information on which it depends becomes available.

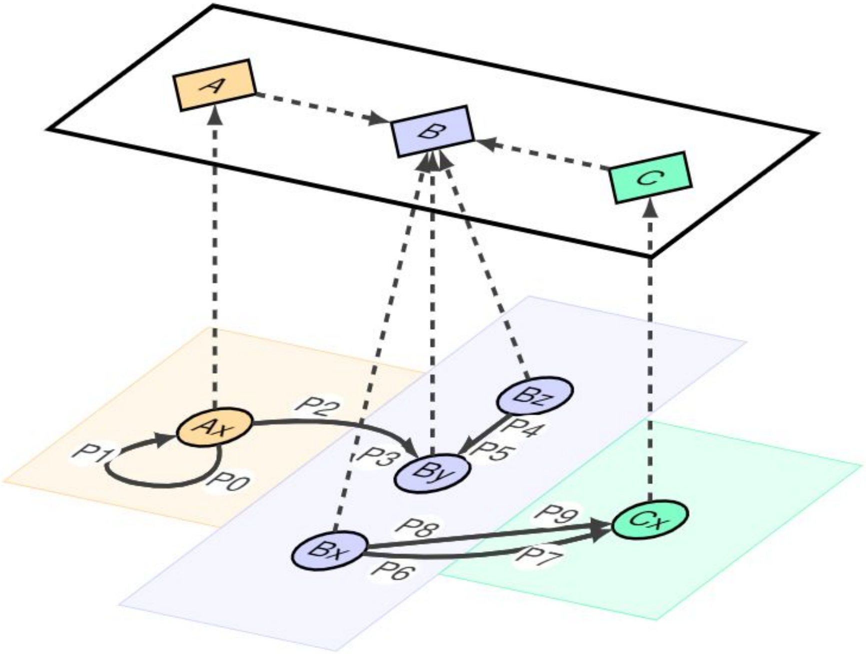


Interactions: Implementation overview

In a nutshell, interactions are realized by two, cooperating, distributed procedures:

1. Component speculation
   The possible behaviors of components are simulated in an encapsulated environment.

2. Solution search & consensus
   Connectors aggregate candidate solution information, ultimately deciding on one.



Part 3_b/4: Internals: Features

Features: Distributed Timeout

The runtime makes a *best effort* to realize a satisfactory interaction, up to a **timeout**.

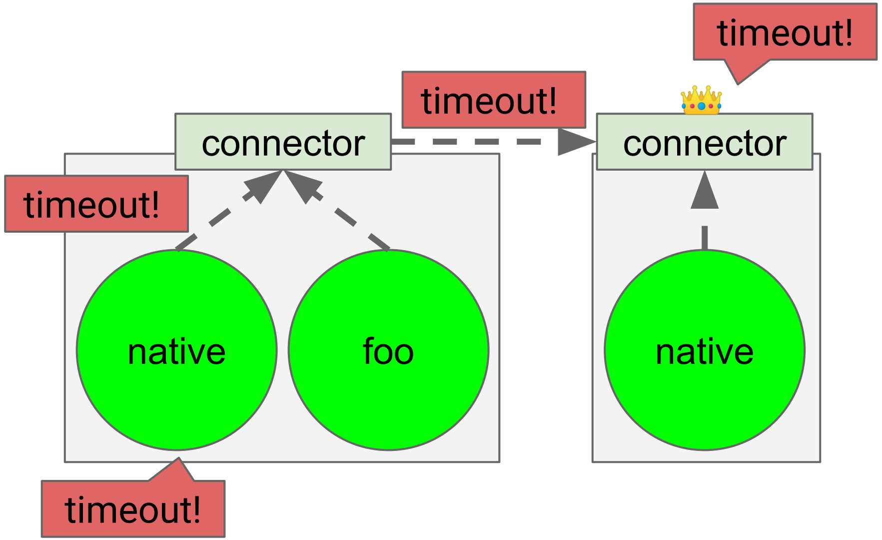
```
connector_sync(c, 100); // 100ms timeout
```

Features: Distributed Timeout

The runtime makes a *best effort* to realize a satisfactory interaction, up to a **timeout**.

```
connector_sync(c, 100); // 100ms timeout
```

A timeout event is consistently observed by all applications, the result of a distributed decision.



Features: Distributed Timeout

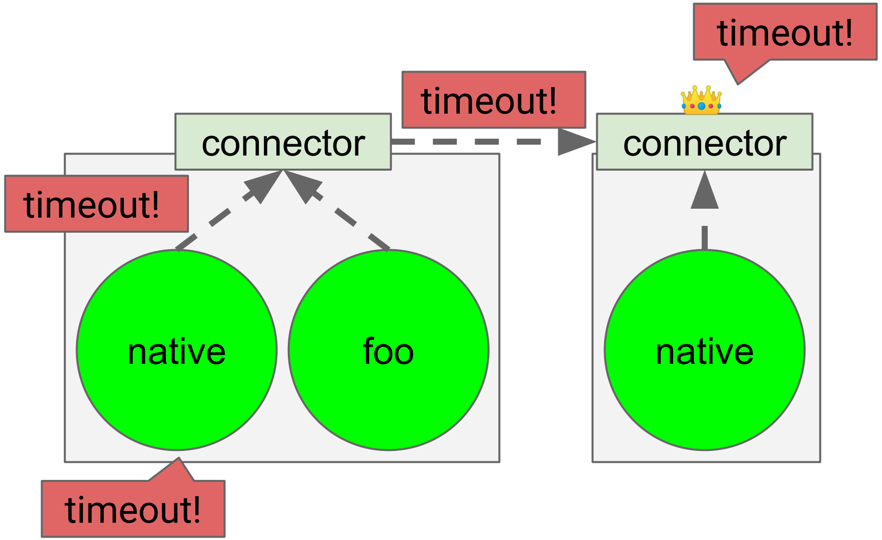
The runtime makes a *best effort* to realize a satisfactory interaction, up to a **timeout**.

```
connector_sync(c, 100); // 100ms timeout
```

A timeout event is consistently observed by all applications, the result of a distributed decision.

Benefits:

- Applications are never starved of control
- The session is always in a consistent state.



Features: Session Transformation

As the session starts, connectors perform **session transformation**, mutating the configuration s.t.:

- The behavior **observable** by native components is unchanged
- Interactions are more efficiently realized

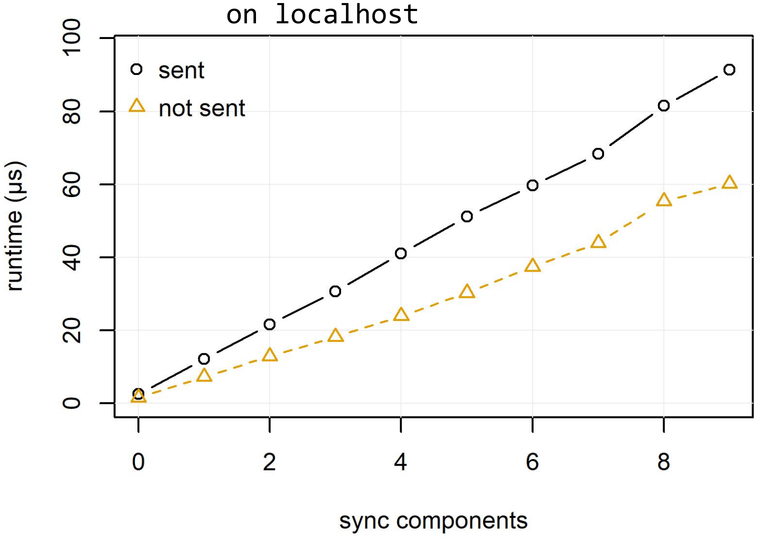
Features: Session Transformation

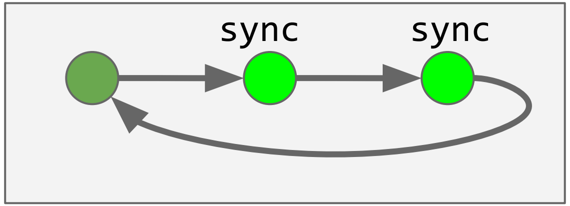
Session transformations can, for example:

...

Features: Session Transformation

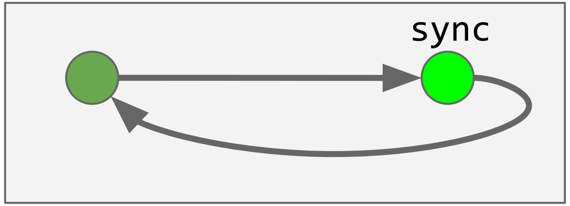
Session transformations can, for example:
Remove idempotent components





before

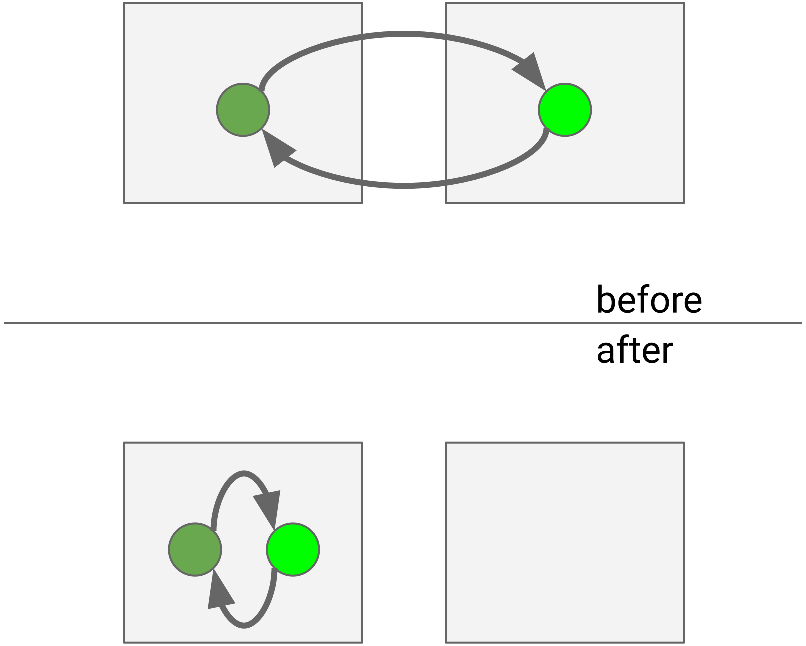
after

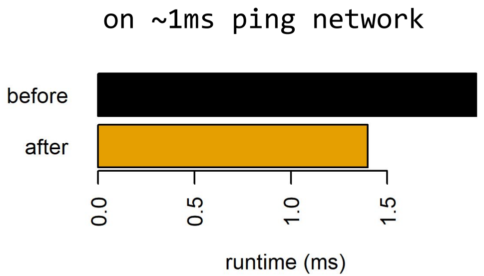


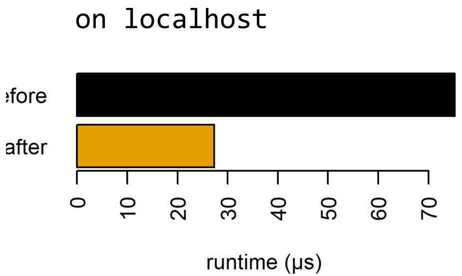
Features: Session Transformation

Session transformations can, for example:

Shorten transport routes



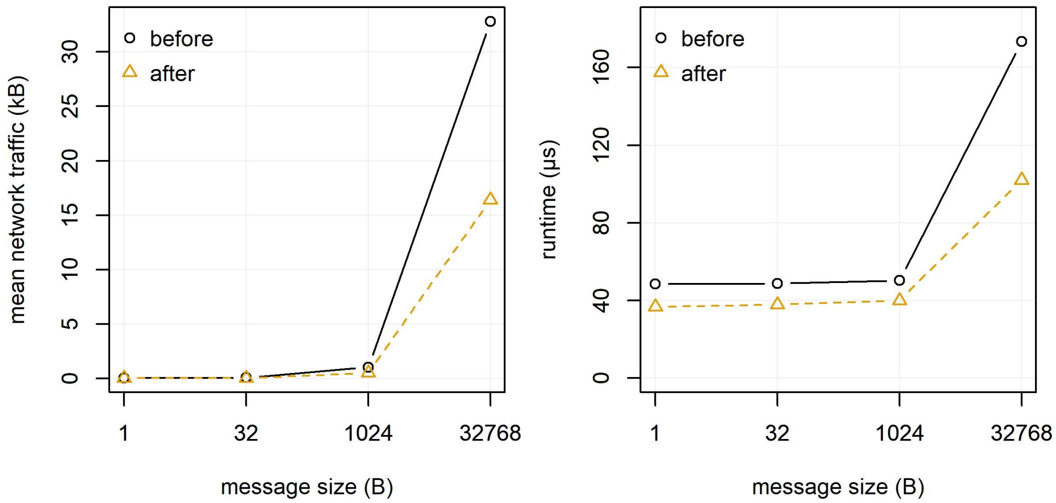


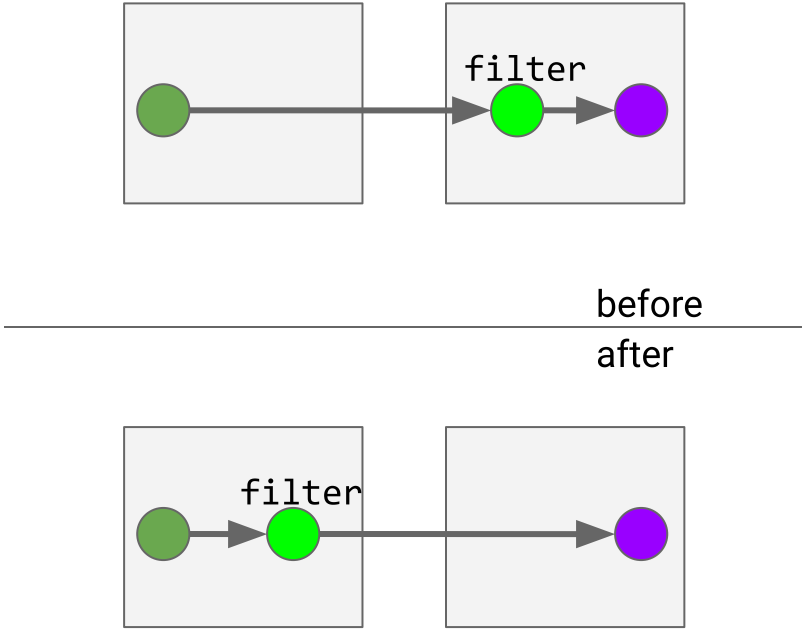


Features: Session Transformation

Session transformations can, for example:
Reduce network traffic

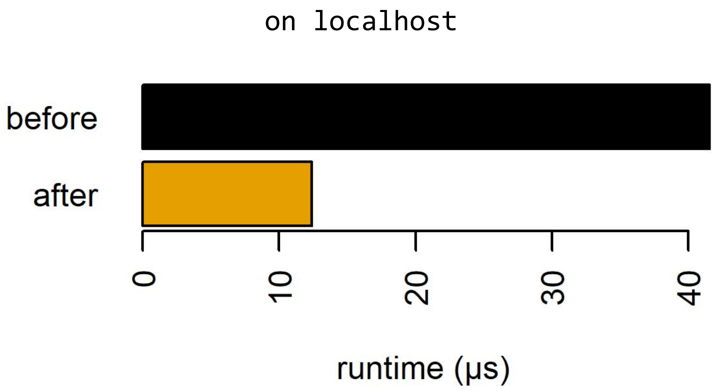
on ~1ms ping network

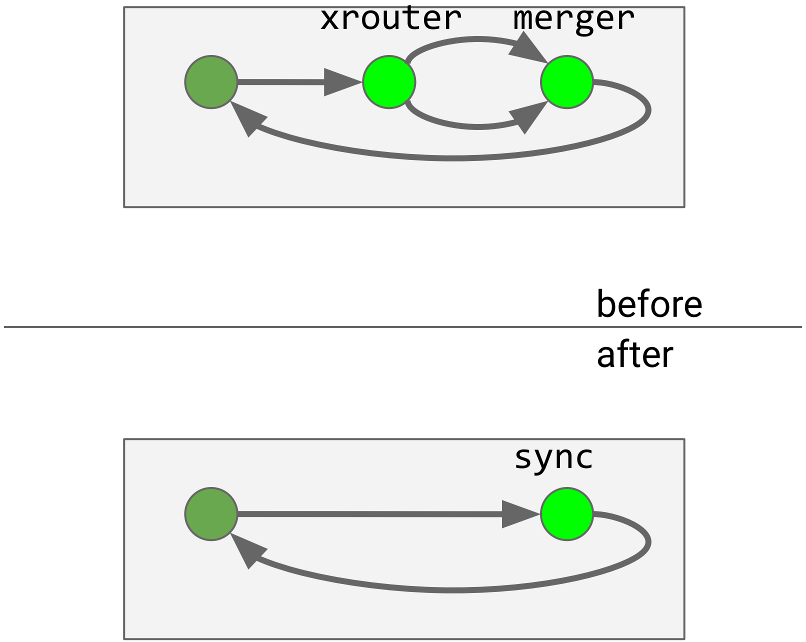




Features: Session Transformation

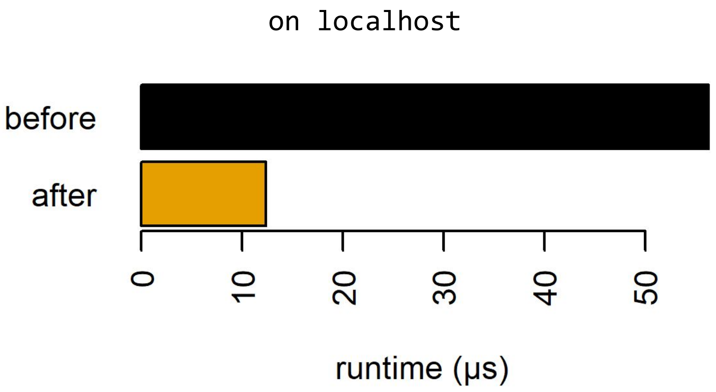
Session transformations can, for example:
Simplify primitive clusters





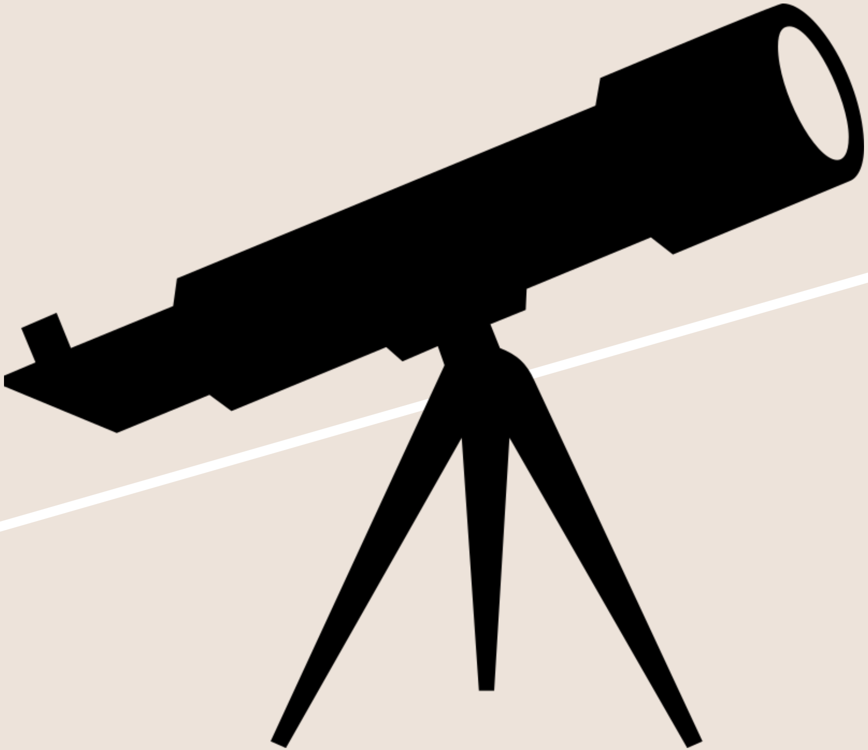
Features: Session Transformation

Session transformations can, for example:
Replace composites with primitives




```
primitive sequencer3(out a, out b, out c) {
    int i = 0;
    while(true) synchronous {
        out to = a;
        if     (i==1) to = b;
        else if(i==2) to = c;
        if(fires(to)) {
            put(to, create(0));
            i = (i + 1)%3;
        }
    }
}
```

Part 4/4:
Future



Future: Improving Flexibility

There are many promising directions for future work that aims to allow the expression of new protocols, or the creation of new kinds of sessions.

1. relax synchronicity to atomicity
2. tree reconfiguration
3. N messages per port per round
4. decouple speculation from decision

Future: Improving Flexibility

1. relax synchronicity to atomicity
2. tree reconfiguration
3. N messages per port per round
4. decouple speculation from decision

```
primitive foo(in a, in b) {
    synchronous { get(a); }
    synchronous { get(b); }
}
```

Consider interactions that advance the state of protocol components any number of ~~synchronous~~ **atomic** blocks (currently 1).

Essentially, takes power away from components and gives it to the runtime.

Advantages:

1. Some component slow $\nRightarrow$ session slow
2. Components become more flexible

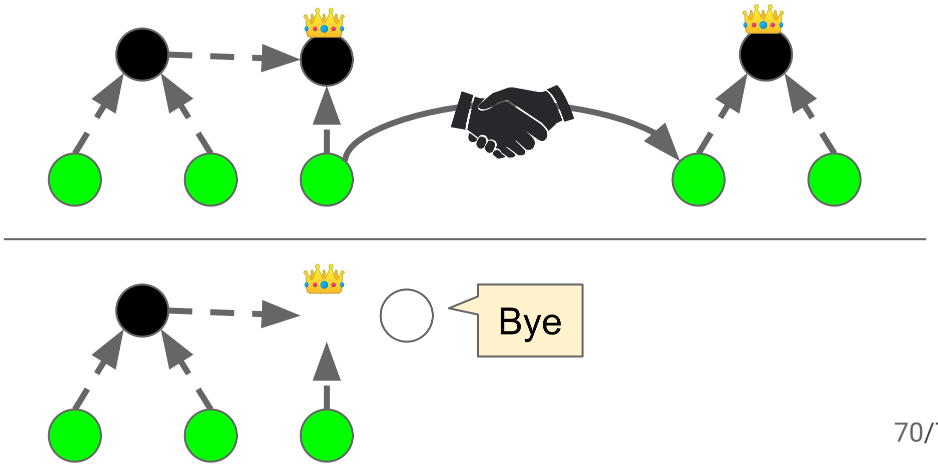
TODO: Need a new system (e.g. priority) to avoid starvation

Future: Improving Flexibility

1. relax synchronicity to atomicity
2. tree reconfiguration
3. N messages per port per round
4. decouple speculation from decision

Solution tree reconfiguration → new capabilities:

1. Dynamic session fusing, splitting
2. Unify setup and communication phases
3. Robustness to control channel breakdown



Future: Improving Flexibility

1. relax synchronicity to atomicity
2. tree reconfiguration
3. N messages per port per round
4. decouple speculation from decision

Currently, components are permitted to send up to 1 message per round. E.g. this results in an error:

```
connector_put_bytes(c, p, "msg 0", 5);
connector_put_bytes(c, p, "msg 1", 5);
connector_sync(c, -1);
```

Future:
Rework implementation, such that sequences of puts, gets are allowed per port, per round. Rework predicates to reason about msg index bounds.

Future: Improving Flexibility

1. relax synchronicity to atomicity
2. tree reconfiguration
3. N messages per port per round
4. decouple speculation from decision

Interactions are found by performing 1-round lookahead, using speculative execution.

Example: This may fail!

```
primitive foo(out a) {
  boolean r0_get;
  synchronous {
    r0_get = fires(a);
    if(r0_get) get(a);
  }
  synchronous { assert(r0_get); }
}
```

Future:
Decouple speculation and decision, such that decisions are made using arbitrary lookahead.

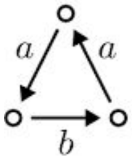
Future: Improving Performance

During development & benchmarking, we identified opportunities for optimizations and restrictions to make connectors more efficient:

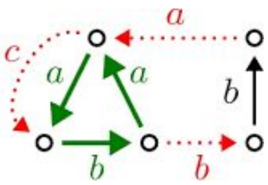
1. Rule-based session transformation
2. Session-wide message aliasing
3. Kernel implementation
4. Control algorithms over UDP/IP

Future: Improving Performance

1. Rule-based session transformation
2. Session-wide message aliasing
3. Kernel implementation
4. Control algorithms over UDP/IP



and



Session transformation is very limited; no support for reasoning about user-defined protocols.

Future:
Session transformation that searches for patterns, based on components' properties ⇒ Transformations more robust, and can work on user-defined protocols.

Example:
Patch graph rewriting (Overbeek & Endrullis) for robust, rule-based session transformations.

Future: Improving Performance

1. Rule-based session transformation
2. Session-wide message aliasing
3. Kernel implementation
4. Control algorithms over UDP/IP

As speculation ⇒ message replication, the runtime has a system for safely aliasing message contents.

As a bonus, messages are cheaply aliased between components with the same manager (connector).

Future:
Decouple message identifiers from message contents. Components exchange and replicate message identifiers primarily.

E.g. Approach: Id(m) = Hash(Contents(m)). Communicate Id and Hash separately, and have connectors populate a local Id⇒Contents store.

Future: Improving Performance

1. Rule-based session transformation
2. Session-wide message aliasing
3. Kernel implementation
4. Control algorithms over UDP/IP

Connectors are implemented in user space.
Boundary to OS is currently between connectors & transport layer.

Future: Implement connector runtime in the kernel.

Benefits:
1. Faster session ⇔ transport interface
2. OS read-only pages for message contents
3. Other OS work can access protocols

Future: Improving Performance

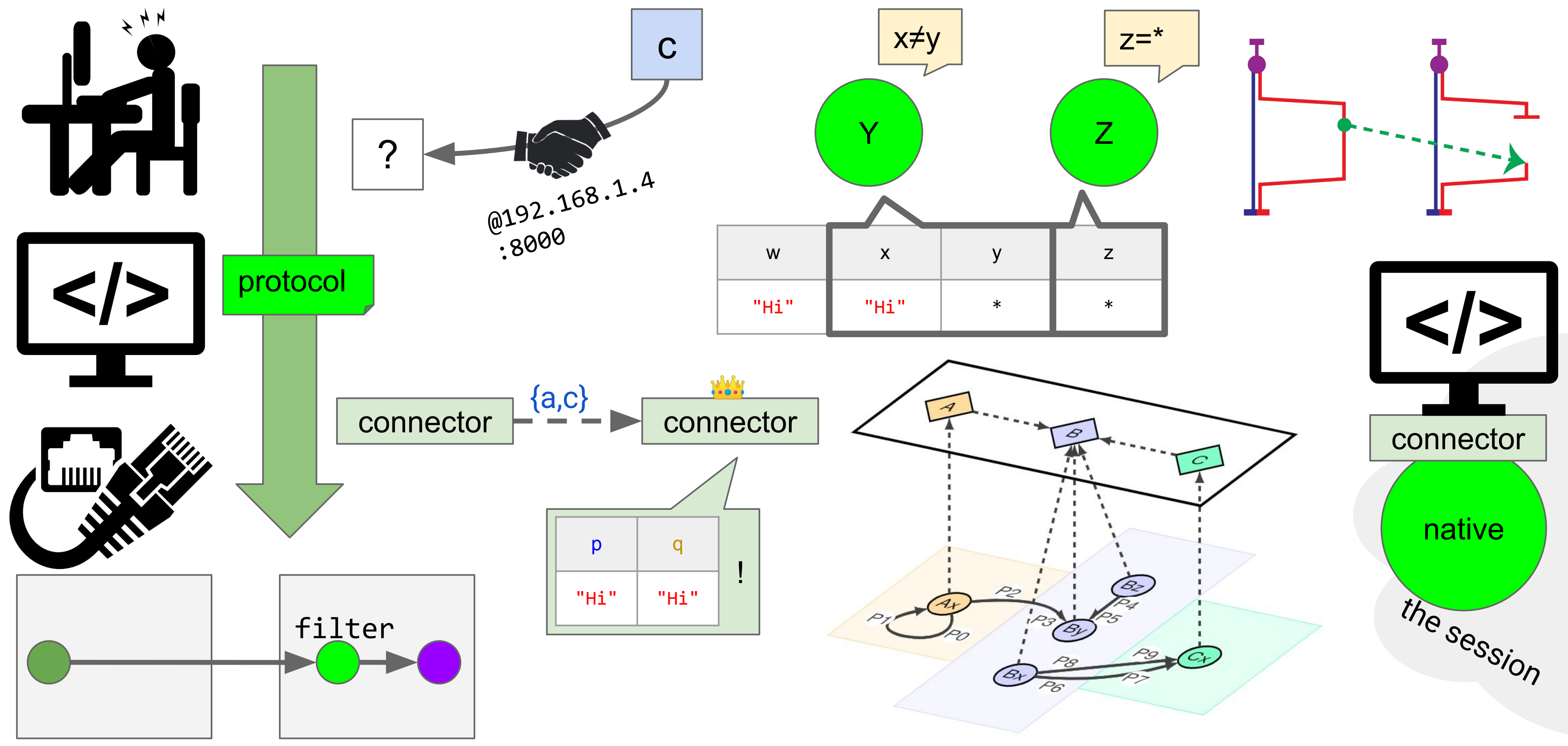
1. Rule-based session transformation
2. Session-wide message aliasing
3. Kernel implementation
4. Control algorithms over UDP/IP

Currently, distributed control messages are transported over TCP. The implementation is simpler as it can rely on TCP for ordering, delivery.

Often, TCP's guarantees are unnecessarily strong.

Future:
Implement control algorithms atop UDP or IP, providing ordering, delivery only as needed.

New control algorithms can be protocol-aware.
Example: retransmit `promising' speculative messages more frequently than others.


</>
protocol
?
c
@192.168.1.4
:8000
x≠y
z=*
Y
Z
w
x
y
z
"Hi"
"Hi"
*
*
connector
{a,c}
connector
p
q
"Hi"
"Hi"
!
filter
A
B
C
Ax
Bx
By
Bz
Cx
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9
</>
connector
native
the session

EXTRA

Future: Improving Flexibility

1. relax synchronicity to atomicity
2. tree reconfiguration
3. N messages per port per round
4. decouple speculation from decision
5. parametric port types

Messages are always byte sequences (~IP packet)

Future:
Make channels,ports generic over a message type.
PDL becomes simpler and safer.
E.g., `in` type becomes `in<[unsigned byte]>`.

```
primitive foo(out<int> o) {
  synchronous {
    put(o, 1234);
  }
}
```

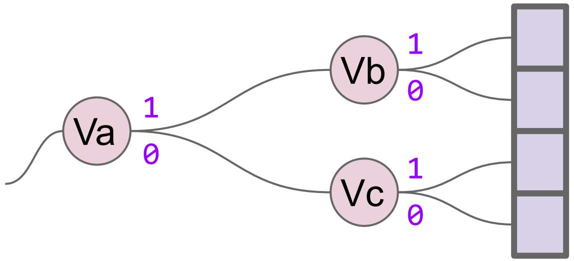
Future: Improving Performance

1. Rule-based session transformation
2. Session-wide message aliasing
3. Kernel implementation
4. Control algorithms over UDP/IP
5. Optimized component storage

To drive speculation, two operations occur often:

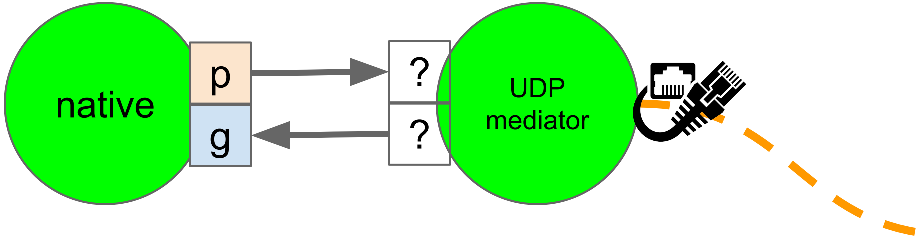
1. Given predicate **P**, visit component branches whose predicates are consistent with **P**.
2. Modify the predicate associated with a component's speculative branch.

Idea: Specialize the storage of branches (currently, `HashMap`) to exploit predicates' structure.



Usage: UDP Interoperability

To facilitate *partial* adoption of connectors, a special **UDP Mediator** component translates between UDP network messages and user payloads, acting much like a native component.




```
Connector * c = connector_new(config);
PortId p, g;
FfiSocketAddr local = {{127, 0, 0, 1}, 7700};
FfiSocketAddr peer  = {{127, 0, 0, 1}, 7701};

connector_add_udp_mediator_component(
  c, &p, &g, local, peer);

connector_connect(c, -1);
```

Q&A: Part 1/4

(1) What kinds of benchmarks would convince someone to use connectors instead of sockets?

We are concerned with two things when comparing applications implemented using sockets/connectors: (1) what can we express, and how safely, easily, and (2) how efficiently does it run?
By comparing the same applications written using sockets and connectors, we could focus on these two aspects; the goal is to show a high gain in expressivity, and low loss of performance (ideally, zero or even negative!). Later, we want to recreate work done for Reo, which resulted in even better runtime performance using Reo. The idea is that, using protocols, the session can perform optimizations that the application could never (safely) do! Eg: slide 64.

(2) Are there properties like privacy that can be formalized now and not with sockets?

We didn't investigate privacy very deeply. We expect such properties aren't *impossible* to formalize either way, but are perhaps more naturally represented using connectors. For example: the runtime reasons about 'locality' of components; we could envision a scheme where users can enforce that the connector will keep an annotated component local to the user's machine (perhaps, because it does sensitive work).

Q&A: Part 2/4

(3) You show session transformations that group multiple components together on a connector. Can you do the opposite? Can you 'spread' a single primitive over the network?

Primitive components are indivisible; however, the work that a primitive performs is often not indivisible. Using a session transformation, we can replace a single primitive component with a cluster (which, together, does the same work), and spread some of these new primitives out over the network.

(4) How do you avoid side-effects when speculatively executing protocol components?

During speculative execution, components can indeed modify their local variables. Implemented incorrectly, this could easily result in one branch's actions leaking to another speculative branch. However, when branching, the local variable stores of branches are forked also, such that each branch has its own isolated workspace.

Q&A: Part 3/4

(5) Can components store large values in their variables? Does branching during speculation not incur too much cost, as a result of these large values being replicated?

Components mostly store small things like integers and bytes, but they can indeed store *messages*, which can be very large. To minimize the cost of replicating messages, the implementation uses a *copy-on-write* pattern, which allows identical messages to be safely aliased between components. Messages are only replicated upon their modification, if they have 2+ aliases. Future work extends this aliasing even further (slide 75).

(6) How do connectors prevent "starving applications of control" (slide 59) if one component can causes all to time out? Can malicious components repeatedly use timeout=0?

Currently, indeed, a malicious component can prevent the session *making progress* by killing the search for interactions before it has a chance to start. However, each time this happens, all applications regain control, and have a chance to change their behavior, including aborting the session.

Currently, each component does indeed have a great influence on the progress of the session, which is quite restrictive. This motivates some of the future work; for example, slide 69 demonstrates a nice way of mitigating the effects of malicious components.

Q&A: Part 4/4

(7) Does PDL have formal semantics? How does it differ from the distributed constraint solving?

We don't have formal semantics yet (28 Oct 2020). This is indeed planned. The idea was for PDL to inherit much of this kind of work done for the Reo language, by PDL differing from Reo as little as possible.