Programming for Autonomy

Amit K. Chopra [1] Munindar P. Singh [2]

[1]Lancaster University

[2]North Carolina State University

16 June 2020 @ PLDI

Autonomy and Systems (20 minutes)

Norms as Meaning (40 minutes)

Protocols (40 minutes)

Protocols in Programming Languages Research (40 minutes)

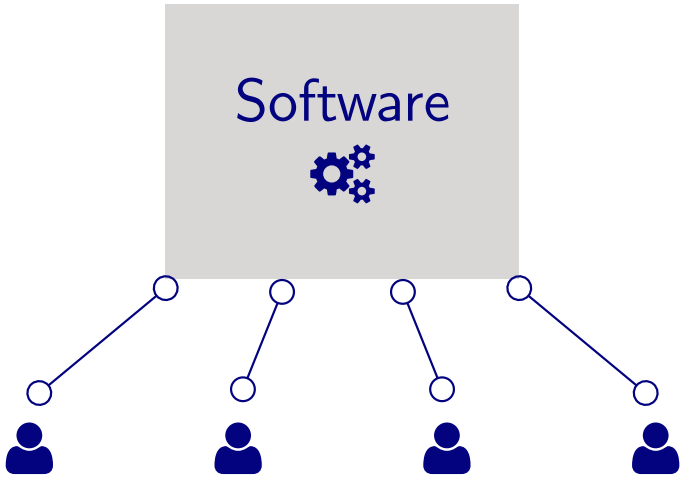
Applying Meanings and Protocols (20 minutes)

Takeaways and Discussion (20)

Outline

Autonomous System as a Central Technical Entity

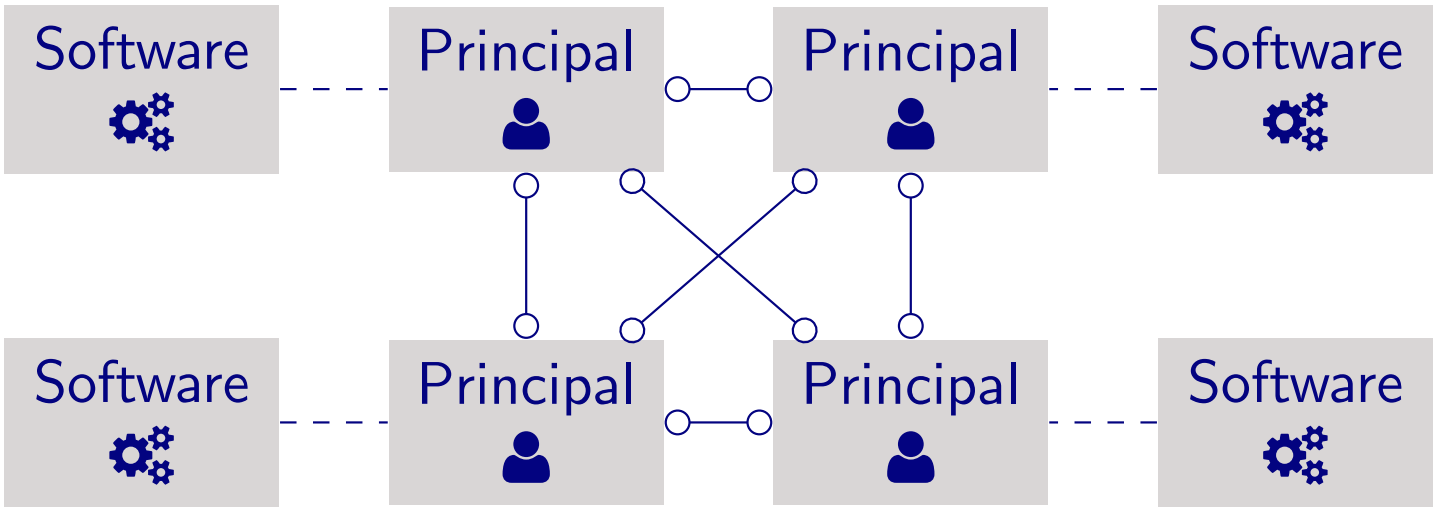
Central technical entity that mediates interactions between users
Such as a prediction algorithm or an autonomous vehicle



- Autonomy is defined as automation: complexity and intelligence
- Software: intelligent agent, service, platform, orchestration,...

Sociotechnical System (STS): System of Autonomous Social Principals

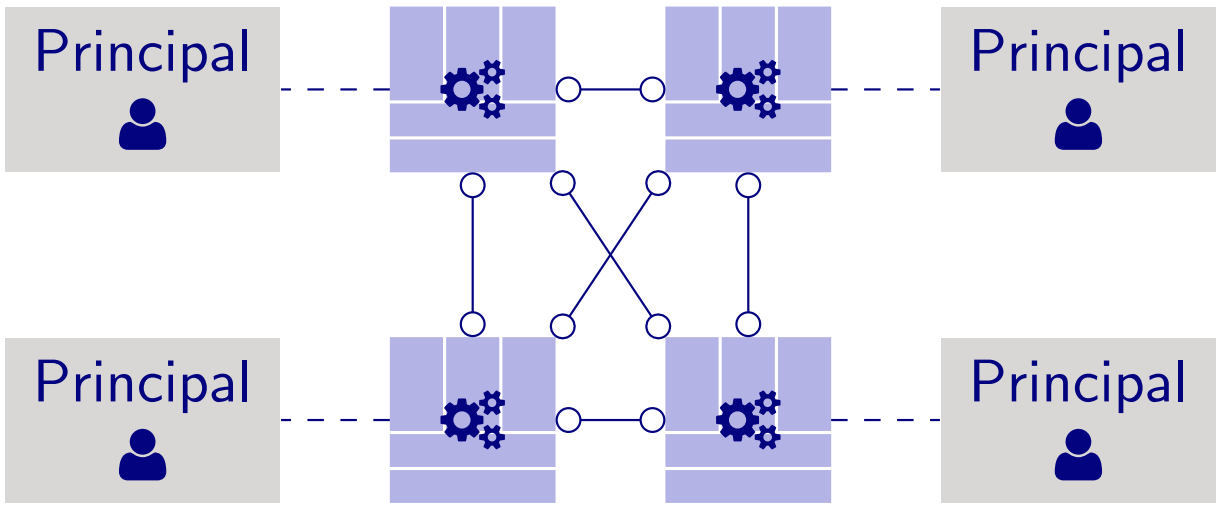
Principals engage on the basis of normative expectations, i.e., norms



- Autonomy as a social construct; mirror of accountability
- Norms
  - Violable, thus accommodating autonomy
  - Directed from accountee to accounter
  - Example: A commitment from Seller to Buyer to deliver upon payment
  - Commonplace: in agreements, regulations, etc.
- Software aids decision making

Agents Helping Principals Exercise Autonomy

Inherently decentralized



- Each agent reflects the autonomy of its principal
- Agents interact at arms-length via asynchronous messaging
- Each agent a locus of state; no other locus of state in system
- How can we realize a multiagent system based that accommodates the autonomy of its principals?

Representing Norms

Social Meaning of Interaction

Commitment

By sending a quote for some goods, Seller commits to Buyer that if payment occurs within five days of quote, then the item will be delivered within ten days of payment

- Representation enables monitoring, for
  - Compliance checking
  - Accountability
  - Decision making
  - System governance
- Declarative representations
- New area of programming language research

Representing the Commitment in Cupid

A view over events, not an execution policy

```
base events
  quote(S, B, ID, item, price)
  pay(S, B, ID, item, amt, addr)
  deliver(S, B, ID, item, addr, status)

commitment PurchaseCom S to B
  create quote
  detach pay within quote + 5d
     where amt >= price
  discharge deliver within pay + 10d
```

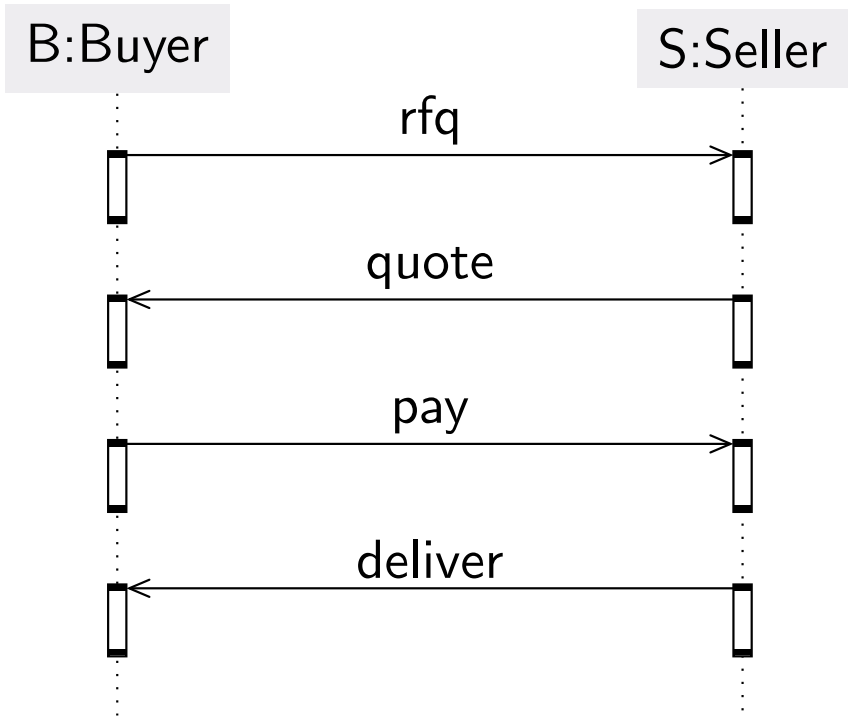
Operationalizing Norms via Protocols

Decentralized computation of norms events

- Who effects an event and when?
- Who observes an event and when?
- Who may generate (bind) a piece of information?

```
base events
  quote(S, B, ID, item, price)
  pay(S, B, ID, item, amt)
  deliver(S, B, ID, item, addr, status)

commitment PurchaseCom S to B
  create quote
  detach pay within quote + 5d
     where amt >= price
  discharge deliver within pay + 10d
```

Protocols as UML Interaction Diagrams



- Informal, synchronous, not compositional, no tooling

Protocols in BSPL

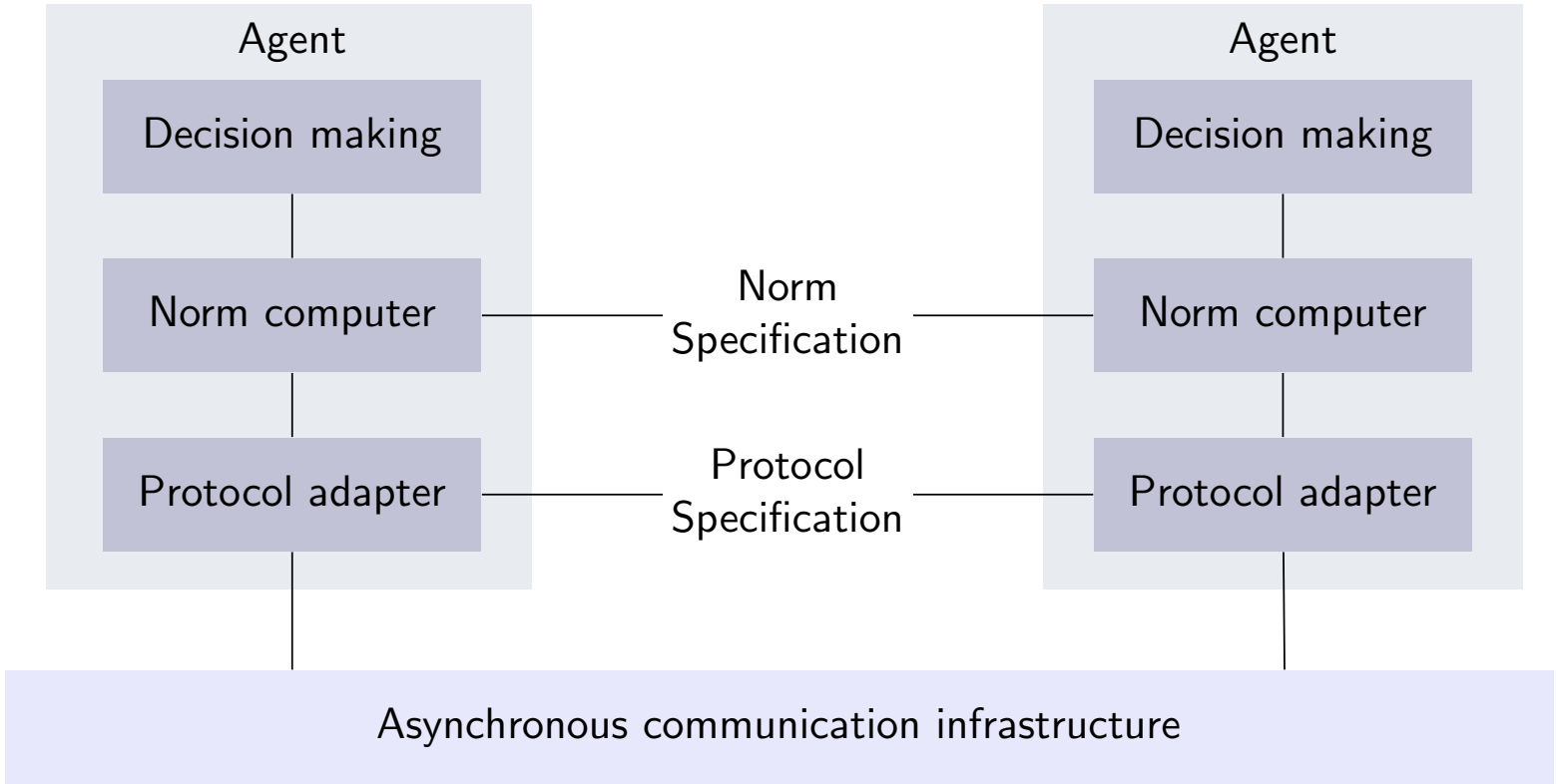
Constrain the occurrence and ordering of messages by specifying information causality and integrity

```
Purchase {
 role Buyer (B), Seller (S)

 parameter out ID key, out item, out addr, out price, out amt, out
      status

 B ↦ S: rfq[out ID, out item, out addr]
 S ↦ B: quote[in ID, in item, out price]
 B ↦ S: pay[in ID, in item, out amt]
 S ↦ B: deliver[in ID, in item, in addr, out status]
}
```

- Superior alternative to session types and trace-based protocol languages

Interaction-Oriented Architecture

Norms and Protocols Model Interactions at Meaning and Operational Levels, Respectively



Asynchronous Communication

- Without message ordering guarantees from the infrastructure (Hewitt and Agha)
- FIFO delivery by infrastructure violates the famous end-to-end argument (Saltzer et al.)
  - Insufficient for application
  - Complexity in infrastructure
  - Imposes cost on applications that don't need it

- Akka and Erlang guarantee FIFO message delivery
- PL approaches for protocols rely on FIFO delivery
- Challenge: Coordinate computation without assuming ordered delivery

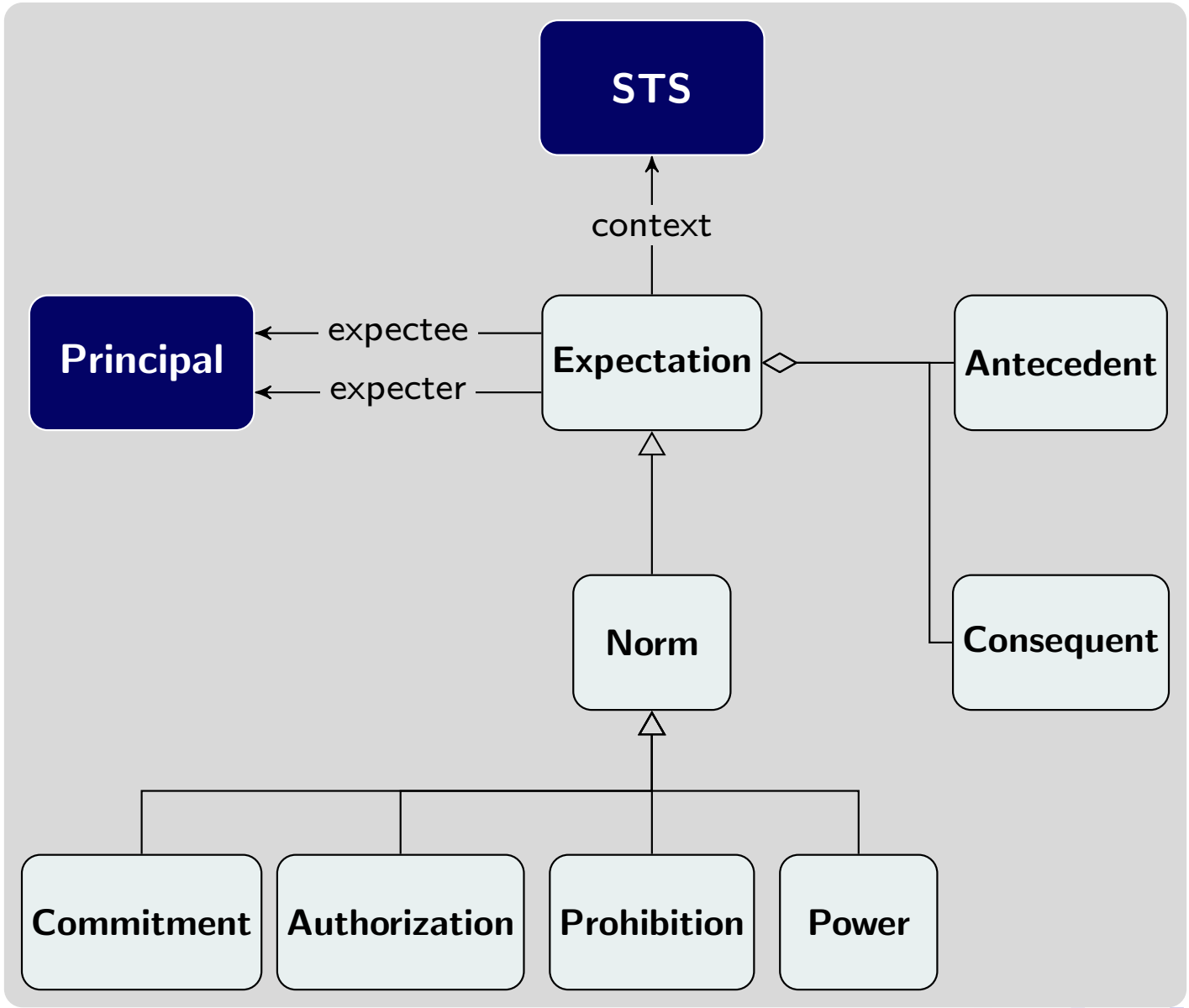
Interaction-Oriented Methodology

Lucid hi-fi computational abstractions for interactions among autonomous principals

- Engineer a system by composing declarative specifications of interactions
  - Strictly without considering agents (endpoint implementations)
- Engineer an agent on the basis of those specifications
  - Strictly without considering other agents

Outline

Approach: Specify Social Expectations as Norms



Norms: Specify Directed Expectations between Roles

Specify the social architecture of an STS

Kinds	**Accountable (Expectee)**	**Privileged (Expecter)**
Commitment	Debtor	Creditor
Prohibition	Prohibitee	Prohibiter
Authorization	Authorizer	Authorizee
Power	Empowering	Empowered

Norms Kinds: Canonical Lifecycles

Captures evolutions of an *instance* of the kind

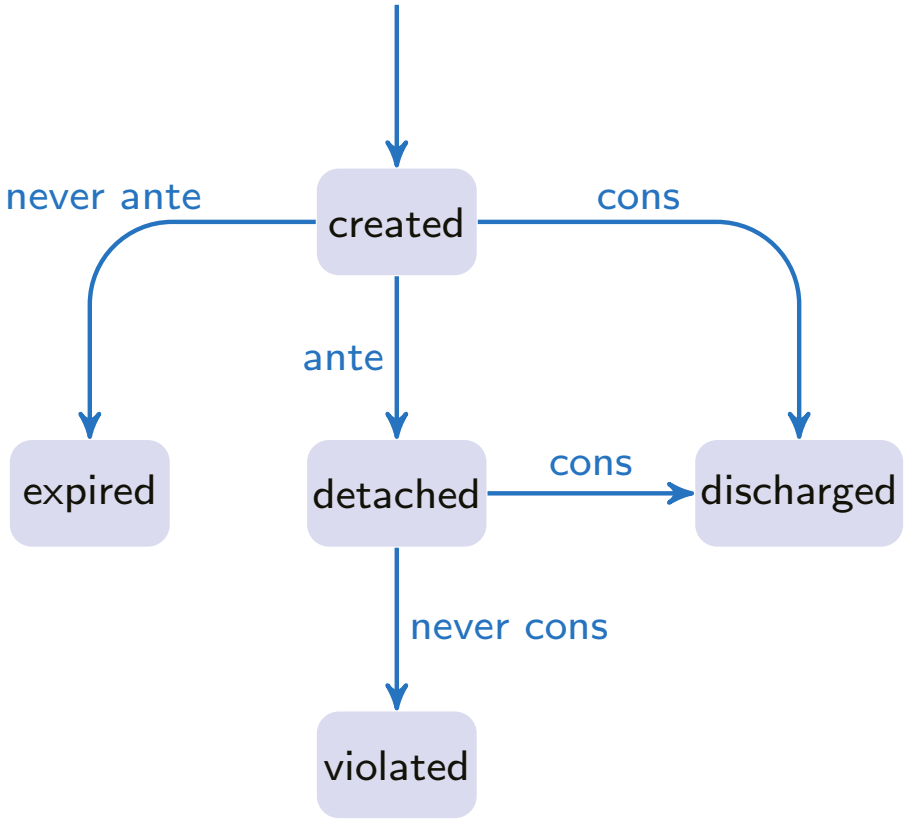
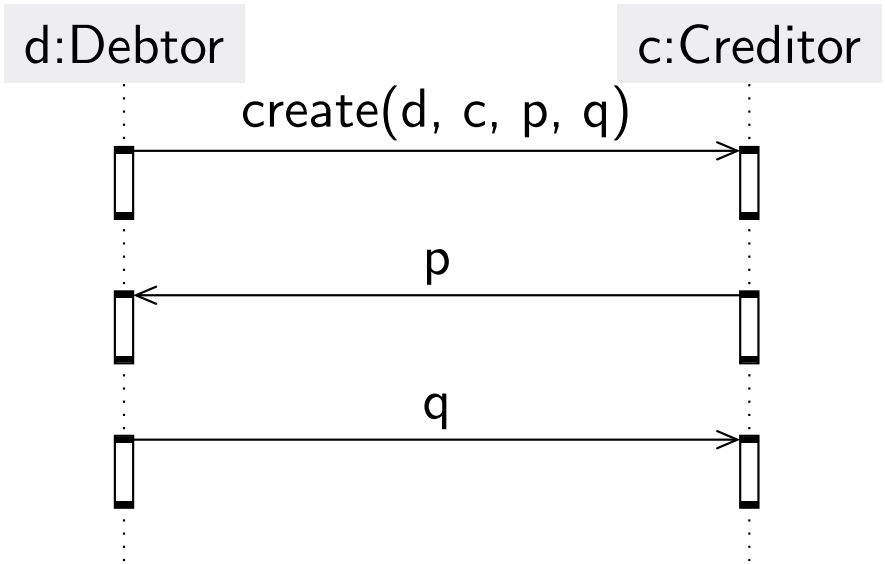


Figure: Commitment lifecycle

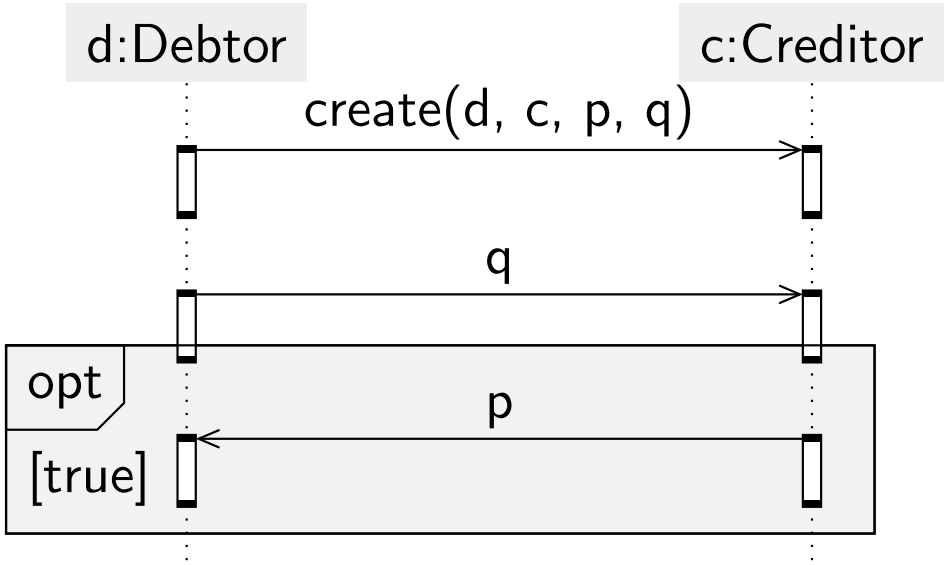
Operationalizing Commitments: Detach then Discharge

C(debtor, creditor, antecedent, consequent)



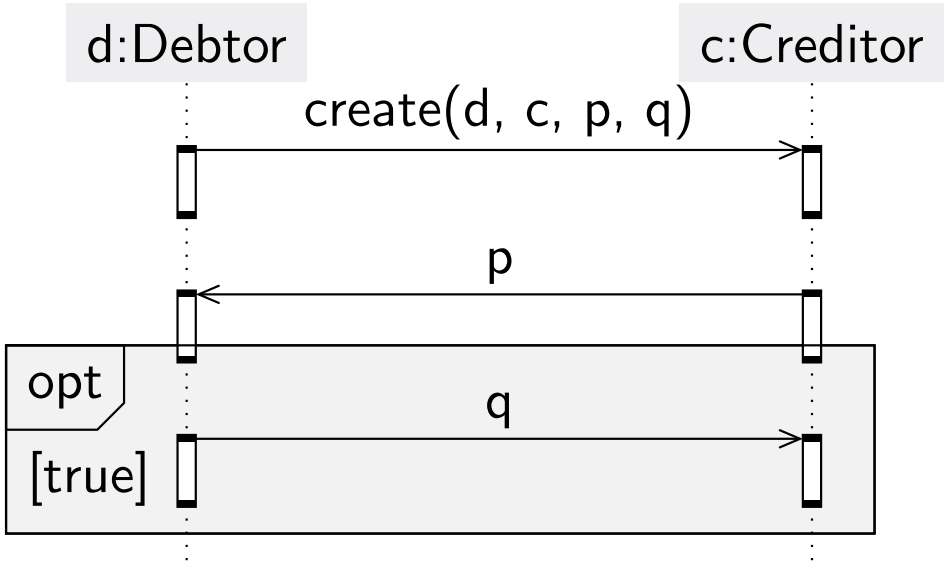
Operationalizing Commitments: Discharge First; Optional Detach

How about this?



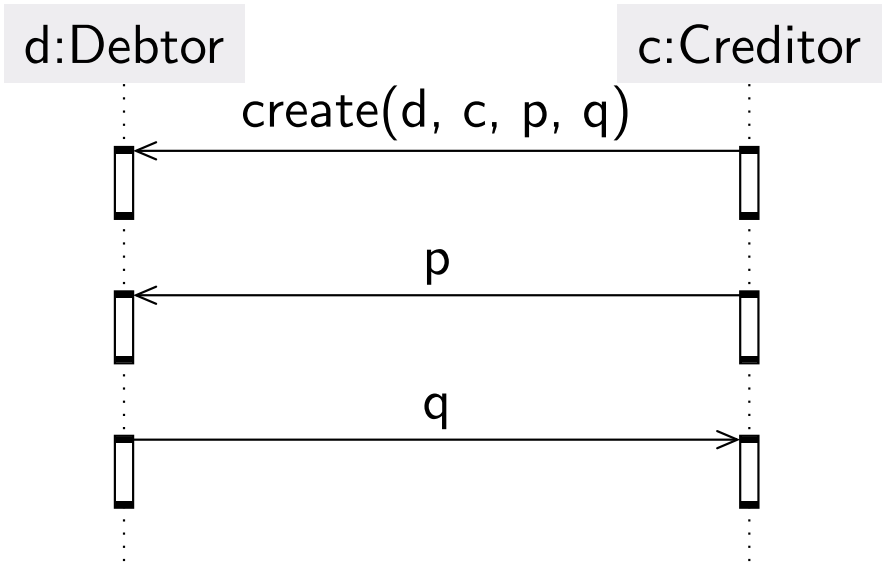
Operationalizing Commitments: Detach First; Optional Discharge

How about this?



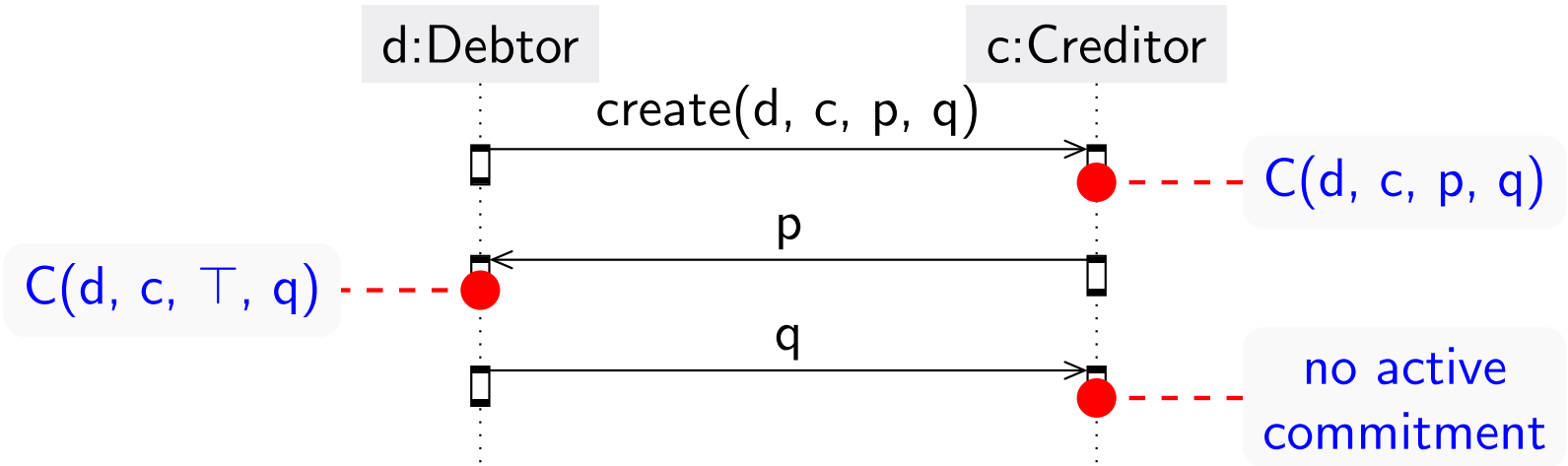
Operationalizing Commitments: Creation by Creditor

C(debtor, creditor, antecedent, consequent)



Operationalizing Commitments: Strengthening by Creditor

C(debtor, creditor, antecedent, consequent)



Norm Specifications as Information Schemas

Technical motivation: Tracking norm instances in information stores

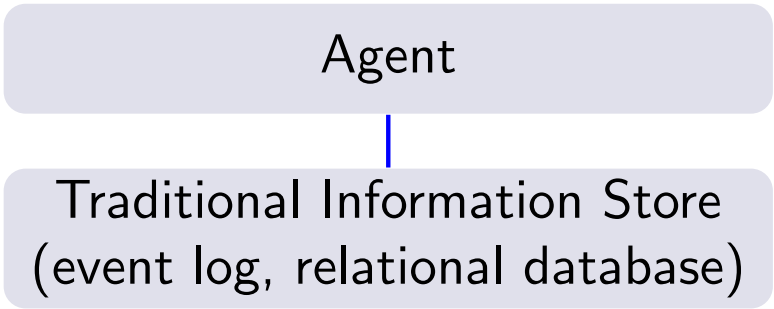


Figure: Existing approaches

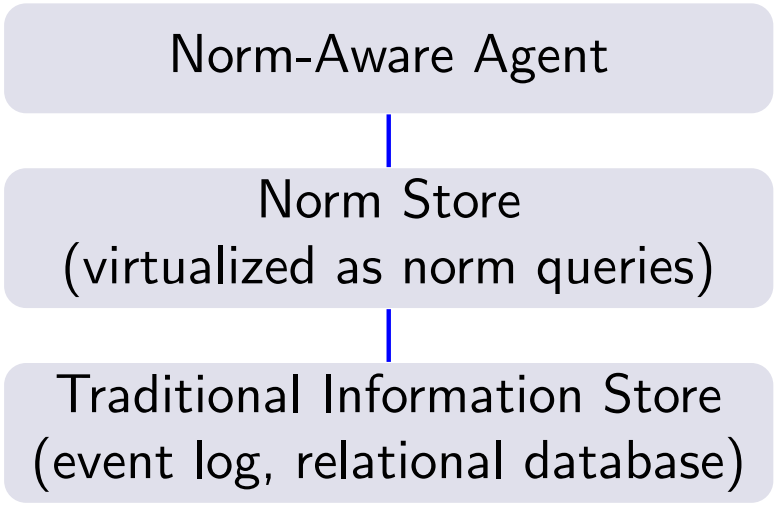


Figure: Cupid

An Information Model and Commitment Specification

E-commerce setting

```
Quote(S, B, ID, item, uPrice, t) with key ID
Accept(B, S, ID, qty, addr, t) with key ID
Payment(B, S, ID, pPrice, t) with key ID
Shipment(S, B, ID, addr, t) with key ID
Refund(S, B, ID, rAmount, t) with key ID

commitment DiscountQuote S to B
 create Quote
 detach Accept[, Quote + 4] and Payment[, Quote + 4]
  where pPrice >= 0.9 * uPrice * qty
 discharge Shipment[, detached DiscountQuote + 4]
```

Canonical Queries for DiscountQuote

- *query-name*(*create-clause*, *detach-clause*, *discharge-clause*)
- Queries for created, detached, expired, discharged, and violated commitment instances
- Implementation produces SQL
- Generated SQL long and complicated; near impossible to write manually
  - violated DiscountQuote is 413 lines long
  - The five queries amount to 1060 lines

Query Results

For times up to 16 June 2020

Quote

ID	item	uPrice	t
T1	fig	1	1 June 2020
T2	pear	1	1 June 2020

Accept

ID	qty	addr	t
T1	1	Lancaster	2 June 2020
T2	1	Raleigh	2 June 2020

Payment

ID	pPrice	t
T1	1	2 June 2020
T2	1	2 June 2020

Shipment

ID	addr	t
T1	Lancaster	3 June 2020

discharged

ID	t
T1	3 June 2020

violated

ID	t
T2	7 June 2020

Example: Compensation

Nested Commitment

```
commitment Compensation S to B
  create Quote
  detach violated(DiscountQuote)
  discharge Refund[, violated(DiscountQuote) + 9] where rAmount =
        pPrice
```

Semantics in Relational Algebra

For a base event E, $[\![E]\!]$ equals its materialized relation. The semantics lifts $[\![\ ]\!]$ to all expressions. A few given below to illustrate the style.

D_1. $[\![E[g,h]]\!] = \sigma_{g \leqslant t < h}[\![E]\!]$. Select all events in E that occur after (including at) g but before h.

D_2. $[\![X \sqcap Y]\!] = \sigma_{t \geq t'}([\![X]\!] \bowtie \rho_{t/t'}[\![Y]\!]) \cup \sigma_{t' < t}(\rho_{t/t'}[\![X]\!] \bowtie [\![Y]\!])$. Select (X, Y) pairs where both have occurred; the timestamp of this composite event is the greater of the two.

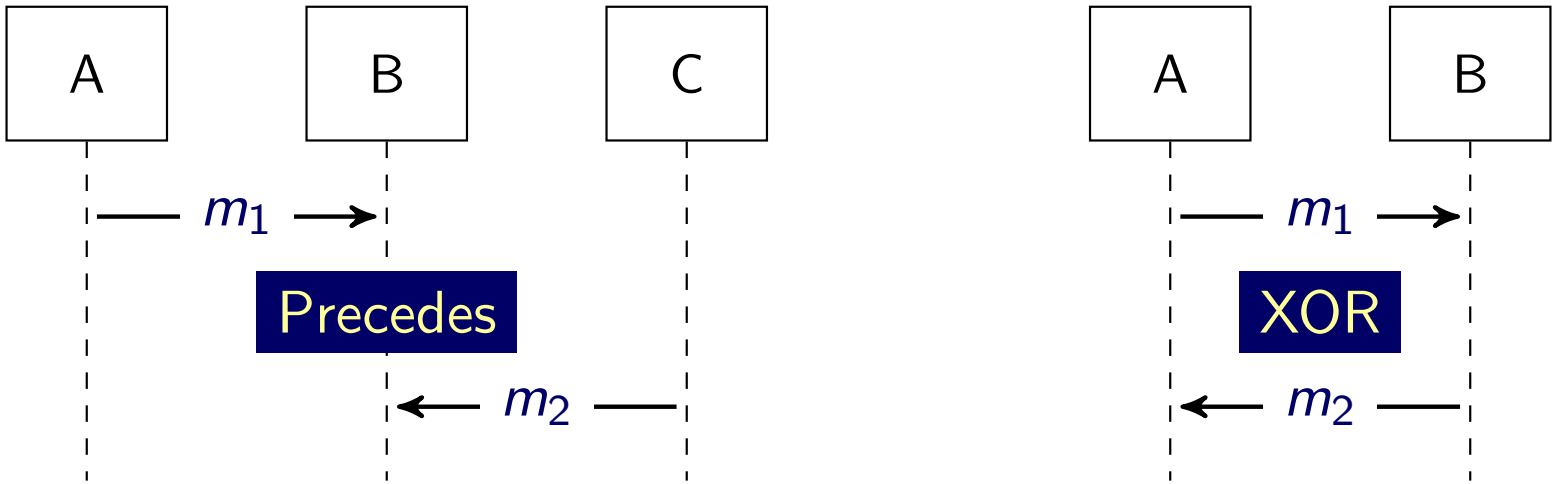
D_3. $[\![\text{created}(c, r, u)]\!] = [\![c]\!]$. A commitment is created when its create event occurs.

D_4. $[\![\text{violated}(c, r, u)]\!] = [\![(c \sqcap r) \ominus u]\!]$. A commitment is violated when it has been created and detached but not discharged within the specified interval.

Outline

Traditional Specifications: Procedural

Low-level, over-specified protocols, easily wrong



- Traditional approaches
  - Emphasize arbitrary ordering and occurrence constraints
  - Then work hard to deal with those constraints
- Our philosophy: The Zen of Distributed Computing
  - Necessary ordering constraints fall out from *causality*
  - Necessary occurrence constraints fall out from *integrity*
  - Unnecessary constraints: simply *ignore* such

Properties of Participants

- Autonomy
- Myopia
  - All choices must be local
  - Correctness must not rely on future interactions
- Heterogeneity: local $\neq$ internal
  - Local state (projection of global state, which is stored nowhere)
    - Public or observable
    - Typically, must be revealed for correctness
  - Internal state
    - Private
    - Must never be revealed: to avoid false coupling
- Shared nothing representation of local state
  - Enact via messaging

BSPL, the Blindingly Simple Protocol Language

Main ideas

- Only *two* syntactic notions
  - Declare a message schema: as an atomic protocol
  - Declare a composite protocol: as a bag of references to protocols
- Parameters are central
  - Provide a basis for expressing meaning in terms of bindings in protocol instances
  - Yield unambiguous specification of compositions through public parameters
  - Capture progression of a role's knowledge
  - Capture the completeness of a protocol enactment
  - Capture uniqueness of enactments through keys
- Separate structure (parameters) from meaning (bindings)
  - Capture many important constraints purely structurally

Key Parameters in BSPL

Marked as ⌜key⌝

- All the key parameters *together* form the key
- Each protocol must define at least one key parameter
- Each message or protocol reference must have at least one key parameter in common with the protocol in whose declaration it occurs
- The key of a protocol provides a basis for the uniqueness of its enactments

Parameter Adornments in BSPL

Capture the essential causal structure of a protocol (for simplicity, assume all parameters are strings)

- ⌜in⌝: Information that must be provided to instantiate a protocol
  - Bindings must exist locally in order to proceed
  - Bindings must be produced through some other protocol
- ⌜out⌝: Information that is generated by the protocol instances
  - Bindings can be fed into other protocols through their ⌜in⌝ parameters, thereby accomplishing composition
  - A standalone protocol must adorn all its public parameters ⌜out⌝
- ⌜nil⌝: Information that is absent from the protocol instance
  - Bindings must not exist

Protocol in BSPL: Main Ideas

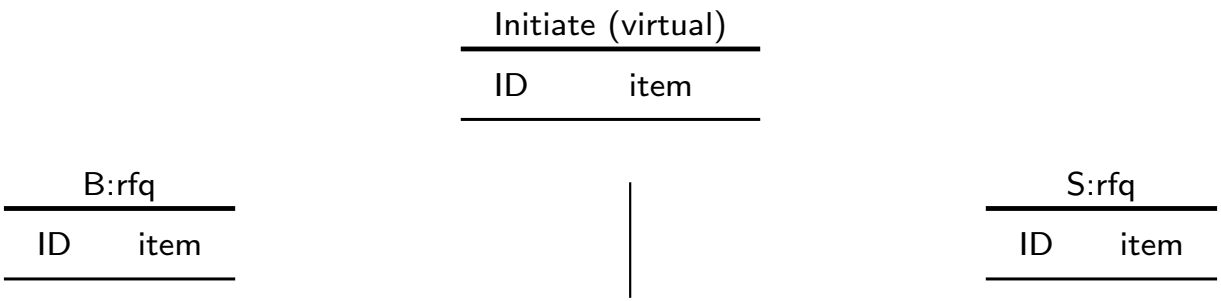
- Declarative
  - No control flow, no control state
- Information-based
  - Specifies the computation of distributed information object
    - Message specification is atomic protocol
  - Specified via parameters
- Explicit causality
  - The messages an agent can send depends upon what it knows
  - Via parameter adornments ⌜out⌝, ⌜in⌝, ⌜nil⌝
- Integrity
  - Agent only sends messages that preserve consistency of objects
  - Via key constraints
- Asynchronous messaging
- Requires no ordering from infrastructure
- Composition and verification

The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```

The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```



The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```

Initiate (virtual)

ID	item
1	fig

B:rfq

ID	item
1	fig

S:rfq

ID	item

The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```

Initiate (virtual)

ID	item
1	fig
5	jam

B:rfq

ID	item
1	fig
5	jam

S:rfq

ID	item

The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```

Initiate (virtual)

ID	item
1	fig
5	jam

B:rfq

ID	item
1	fig
5	jam

S:rfq

ID	item
5	jam

The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```

Initiate (virtual)

ID	item
1	fig
5	jam

B:rfq

ID	item
1	fig
5	jam
×1	apple

S:rfq

ID	item
5	jam

The *Initiate* protocol

```
Initiate {
 role B, S
 parameter out ID key, out item

 B ↦ S: rfq[out ID key, out item]
}
```

Initiate (virtual)

ID	item
1	fig
5	jam
8	fig

B:rfq

ID	item
1	fig
5	jam
8	fig

S:rfq

ID	item
5	jam

The *Offer* Protocol

```
Offer {
 role B, S
 parameter out ID key, out item, out price

 B ↦ S: rfq[out ID, out item]
 S ↦ B: quote[in ID, in item, out price]
}
```

The *Offer* Protocol

```
Offer {
 role B, S
 parameter out ID key, out item, out price

 B ↦ S: rfq[out ID, out item]
 S ↦ B: quote[in ID, in item, out price]
}
```

Offer (virtual)

ID	item	price
1	fig	

B:rfq

ID	item
1	fig

B:quote

ID	item	price

S:rfq

ID	item
1	fig

S:quote

ID	item	price

The *Offer* Protocol

```
Offer {
 role B, S
 parameter out ID key, out item, out price

 B ↦ S: rfq[out ID, out item]
 S ↦ B: quote[in ID, in item, out price]
}
```

Offer (virtual)

ID	item	price
1	fig	10

B:rfq

ID	item
1	fig

B:quote

ID	item	price

S:rfq

ID	item
1	fig

S:quote

ID	item	price
1	fig	10

The *Offer* Protocol

```
Offer {
 role B, S
 parameter out ID key, out item, out price

 B ↦ S: rfq[out ID, out item]
 S ↦ B: quote[in ID, in item, out price]
}
```

Offer (virtual)

ID	item	price
1	fig	10

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

S:rfq

ID	item
1	fig

S:quote

ID	item	price
1	fig	10

The *Offer* Protocol

```
Offer {
 role B, S
 parameter out ID key, out item, out price

 B ↦ S: rfq[out ID, out item]
 S ↦ B: quote[in ID, in item, out price]
}
```

Offer (virtual)

ID	item	price
1	fig	10

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

S:rfq

ID	item
1	fig

S:quote

ID	item	price
1	fig	10
×4	fig	10

The *Decide Offer* Protocol

Choice: *accept* and a *reject* with the same ID *cannot* both occur

```
Decide Offer {
 role B, S
 parameter out ID key, out item, out price, out decision

 B ↦ S: rfq[out ID, out item]
 S ↦ B: quote[in ID, in item, out price]

 B ↦ S: accept[in ID, in item, in price, out decision]
 B ↦ S: reject[in ID, in item, in price, out decision]
}
```

The *Decide Offer* Protocol

Choice: *accept* and a *reject* with the same ID *cannot* both occur

```
Decide Offer {
 role B, S
 parameter out ID key, out item, out price, out decision

 B ↦ S: rfq[out ID, out item]
 S ↦ B: quote[in ID, in item, out price]

 B ↦ S: accept[in ID, in item, in price, out decision]
 B ↦ S: reject[in ID, in item, in price, out decision]
}
```

Decide Offer (virtual)

ID	item	price	decision
1	fig	10	

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

B:accept

ID	item	price	decision

B:reject

ID	item	price	decision

The *Decide Offer* Protocol

Choice: *accept* and a *reject* with the same ID *cannot* both occur

```
Decide Offer {
 role B, S
 parameter out ID key, out item, out price, out decision

 B ↦ S: rfq[out ID, out item]
 S ↦ B: quote[in ID, in item, out price]

 B ↦ S: accept[in ID, in item, in price, out decision]
 B ↦ S: reject[in ID, in item, in price, out decision]
}
```

Decide Offer (virtual)

ID	item	price	decision
1	fig	10	nice

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

B:accept

ID	item	price	decision
1	fig	10	nice

B:reject

ID	item	price	decision

The *Decide Offer* Protocol

Choice: *accept* and a *reject* with the same ID *cannot* both occur

```
Decide Offer {
 role B, S
 parameter out ID key, out item, out price, out decision

 B ↦ S: rfq[out ID, out item]
 S ↦ B: quote[in ID, in item, out price]

 B ↦ S: accept[in ID, in item, in price, out decision]
 B ↦ S: reject[in ID, in item, in price, out decision]
}
```

Decide Offer (virtual)

ID	item	price	decision
1	fig	10	nice

B:rfq

ID	item
1	fig

B:quote

ID	item	price
1	fig	10

B:accept

ID	item	price	decision
1	fig	10	nice

B:reject

ID	item	price	decision
×1	fig	10	nice

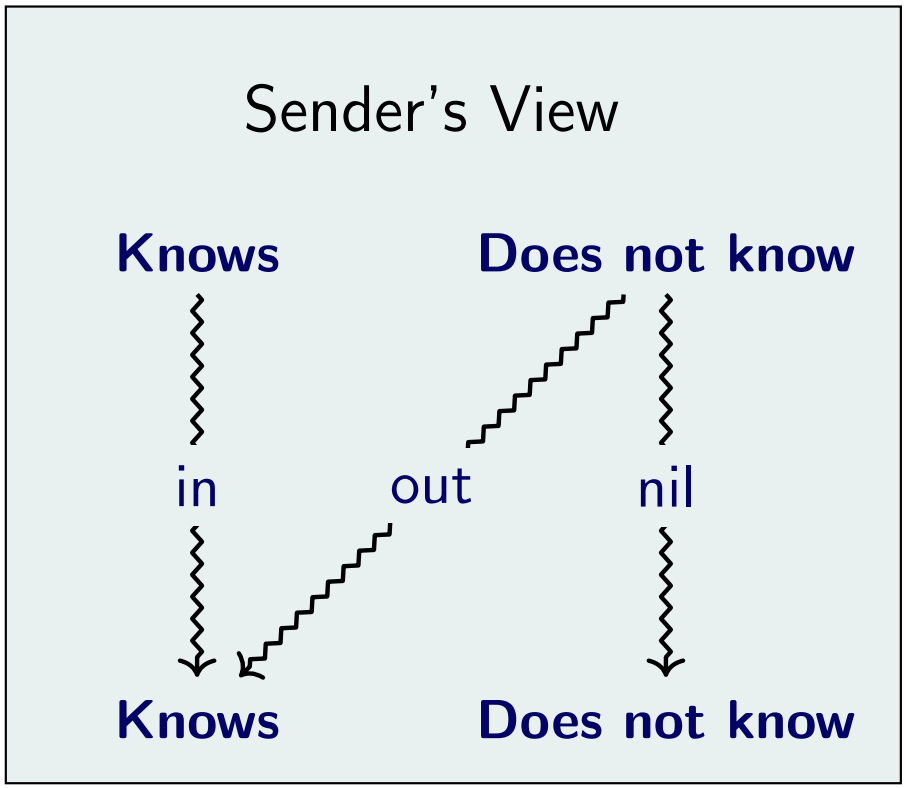
The *Purchase* Protocol

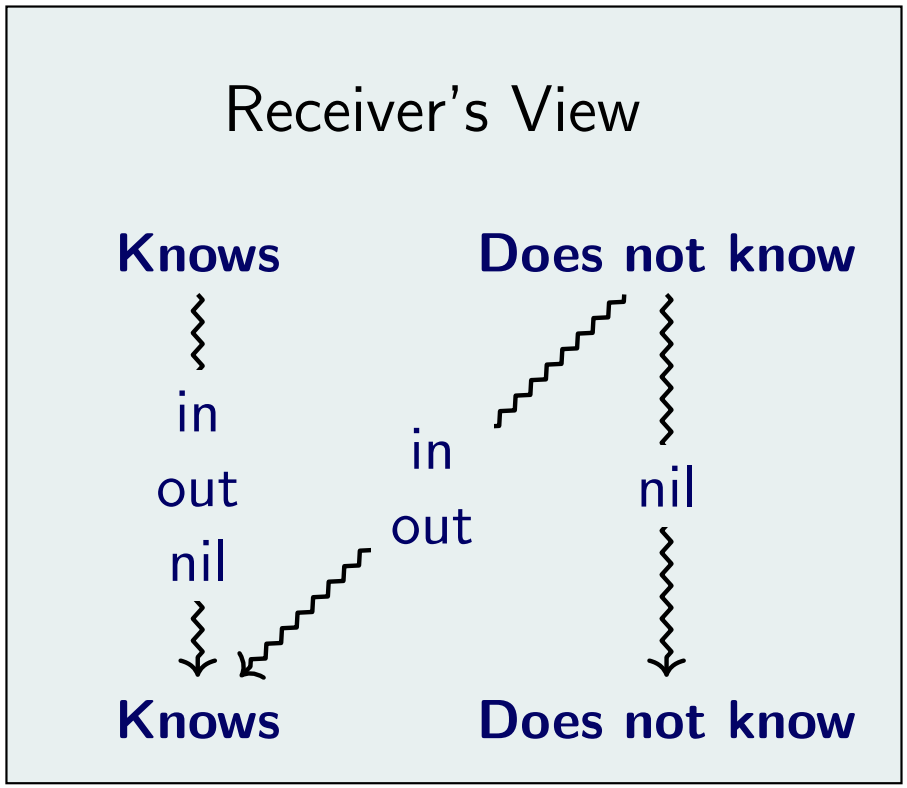
```
Purchase {
 role B, S, Shipper
 parameter out ID key, out item, out price, out outcome
 private address, resp

 B ↦ S: rfq[out ID, out item]
 S ↦ B: quote[in ID, in item, out price]
 B ↦ S: accept[in ID, in item, in price, out address, out resp]
 B ↦ S: reject[in ID, in item, in price, out outcome, out resp]

 S ↦ Shipper: ship[in ID, in item, in address]
 Shipper ↦ B: deliver[in ID, in item, in address, out outcome]
}
```

- *reject* conflicts with *accept* on resp (a *private* parameter)
- *reject* or *deliver* must occur for completion (to bind outcome)

Knowledge and Viability

When is a message viable? What effect does it have on a role's local knowledge?

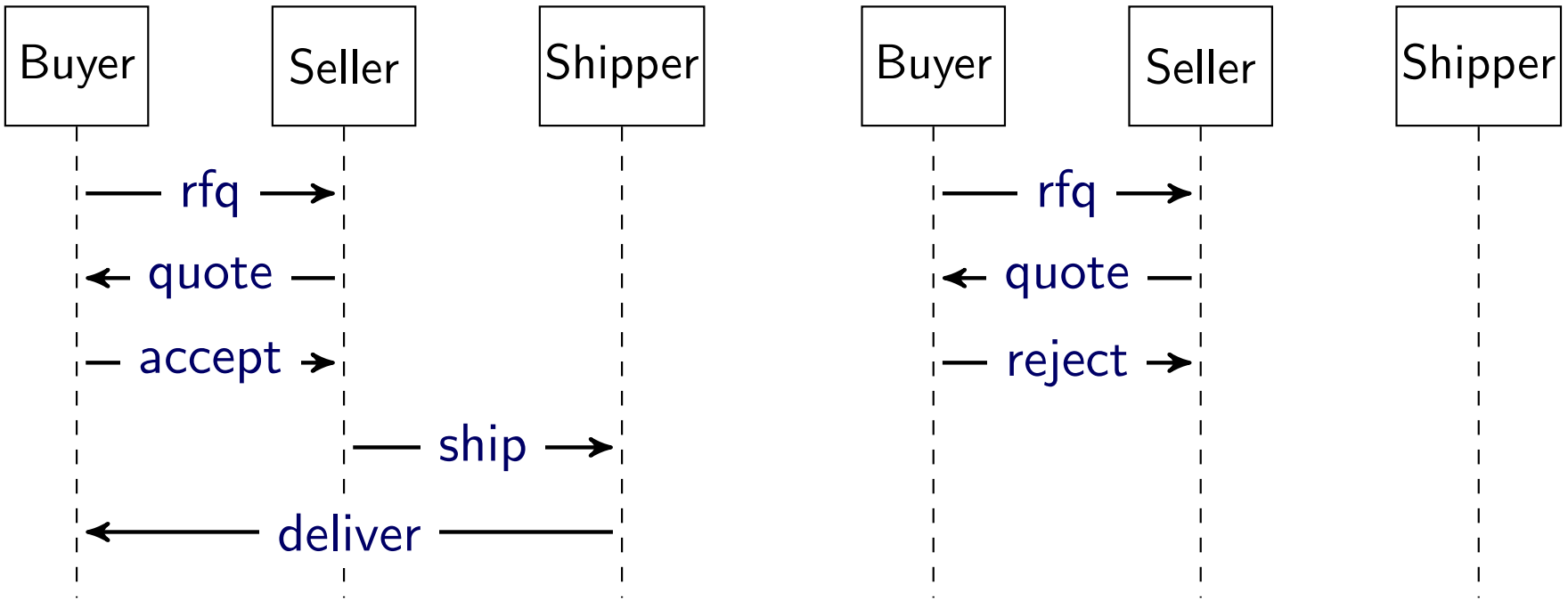




- Knowledge increases monotonically at each role
- An ⌜out⌝ parameter **creates** and transmits knowledge
- An ⌜in⌝ parameter transmits knowledge
- Repetitions through multiple paths are harmless and superfluous

Possible Enactments as Sets of Local Histories

Each participant's local history: set of messages sent and received



Standing Order

As in insurance claims processing

```
Insurance-Claims {
 role Vendor (V), Subscriber (S)
 parameter out pID key, out cID key, out claim, out response

 Create-Policy(V, S, out pID, out details)
 S ↦ V: claimRequest[in pID, out cID, out claim]
 V ↦ S: claimResponse[in pID, in cID, out response]
}
```

- Illustrates composite keys
  - A policy (identified by a binding for pID) may be associated with multiple claims (each identified by a binding for cID)
- Composes protocol Create-Policy, which produces bindings for pID

in-out Polymorphism

price could be ⌜in⌝ or ⌜out⌝

```
Flexible-Offer {
 role B, S
 parameter in ID key, out item, price, out qID

 B ↦ S: rfq[in ID, out item, nil price]
 B ↦ S: rfq[in ID, out item, in price]

 S ↦ B: quote[in ID, in item, out price, out qID]
 S ↦ B: quote[in ID, in item, in price, out qID]
}
```

- The price can be adorned ⌜in⌝ or ⌜out⌝ in a reference to this protocol

Flexible Sourcing of out Parameters

Buyer or Seller Offer

```
Buyer-or-Seller-Offer {
 role Buyer, Seller
 parameter in ID key, out item, out price, out confirmed

 Buyer ↦ Seller: rfq[in ID, out item, nil price]
 Buyer ↦ Seller: rfq[in ID, out item, out price]

 Seller ↦ Buyer: quote[in ID, in item, out price, out
     confirmed]
 Seller ↦ Buyer: quote[in ID, in item, in price, out confirmed]
}
```

- The BUYER or the SELLER may determine the binding
- The BUYER has first dibs in this example

Remark on Control versus Information Flow

- Control flow
  - Natural within a single computational thread
  - Exemplified by conditional branching
  - Presumes master-slave relationship across threads
  - Impossible between mutually autonomous parties because neither controls the other
  - May sound appropriate, but only because of long habit
- Information flow
  - Natural across computational threads
  - Explicitly tied to causality

Summary: Main Ideas

Taking a declarative, information-centric view of interaction to the limit

- Specification
  - A message is an atomic protocol
  - A composite protocol is a set of references to protocols
  - Each protocol is given by a name and a set of parameters (including keys)
  - Each protocol has *inputs* and *outputs*
- Representation
  - A protocol corresponds to a relation (table)
  - Integrity constraints apply on the relations
- Enactment via LoST: Local State Transfer
  - Information represented: local $\neq$ internal
  - Purely decentralized at each role
  - Materialize the relations *only* for messages

Realizing BSPL via LoST

Think of the message logs you want

- For each role
  - For each message that it sends or receives
    - Maintain a *local* relation of the same schema as the message
- Receive and store any message provided
  - It is not a duplicate
  - Its integrity checks with respect to parameter bindings
  - Garbage collect expired sessions: requires additional annotations
- Send any unique message provided
  - Parameter bindings agree with previous bindings for the same keys for ⌜in⌝ parameters
  - No bindings for ⌜out⌝ and ⌜nil⌝ parameters exist

Information Centrism

Characterize each interaction purely in terms of information

- Explicit causality
  - Flow of information coincides with flow of causality
  - No hidden control flows
  - No backchannel for coordination
- Keys
  - Uniqueness
  - Basis for completion
- Integrity
  - Parameter has only one value (relative to its value of its key)
- Immutability
  - Durability
  - Robustness: insensitivity to
    - Reordering by infrastructure
    - Retransmission: one delivery is all it needs

Safety: *Purchase Unsafe*

Remove conflict between *accept* and *reject*

```
Purchase Unsafe {
 role B, S, Shipper
 parameter out ID key, out item, out price, out outcome
 private address, resp

 B ↦ S: rfq[out ID, out item]
 S ↦ B: quote[in ID, in item, out price]
 B ↦ S: accept[in ID, in item, in price, out address]
 B ↦ S: reject[in ID, in item, in price, out outcome]

 S ↦ Shipper: ship[in ID, in item, in address]
 Shipper ↦ B: deliver[in ID, in item, in address, out outcome]
}
```

- B can send both *accept* and *reject*
- Thus outcome can be bound twice in the same enactment

Liveness: *Purchase No Ship*

Omit *ship*

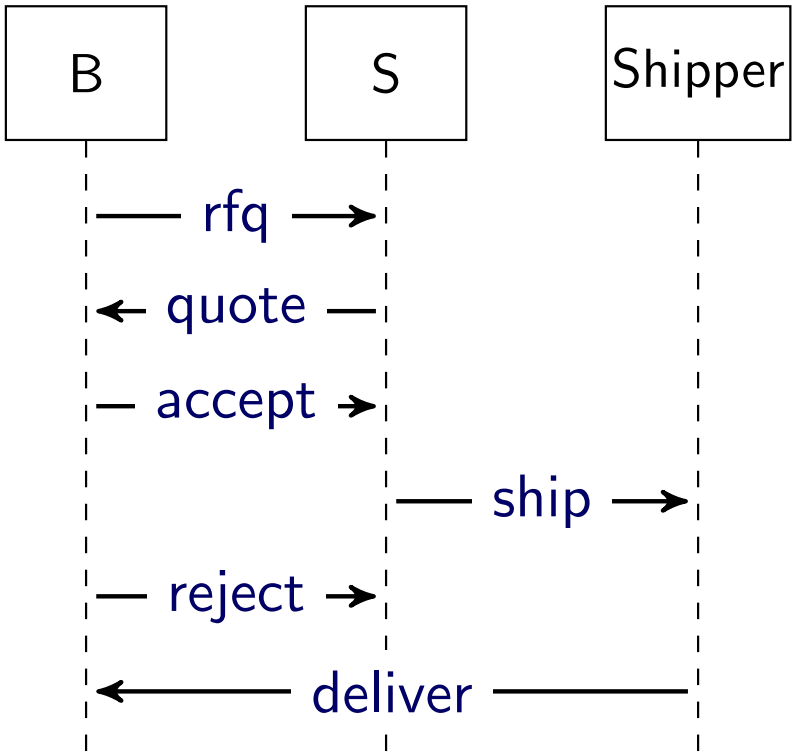
```
Purchase No Ship {
 role B, S, Shipper
 parameter out ID key, out item, out price, out outcome
 private address, resp

 B ↦ S: rfq[out ID, out item]
 S ↦ B: quote[in ID, in item, out price]
 B ↦ S: accept[in ID, in item, in price, out address, out resp]
 B ↦ S: reject[in ID, in item, in price, out outcome, out resp]

 Shipper ↦ B: deliver[in ID, in item, in address, out outcome]
}
```

- If B sends *reject*, the enactment completes
- If B sends *accept*, the enactment deadlocks

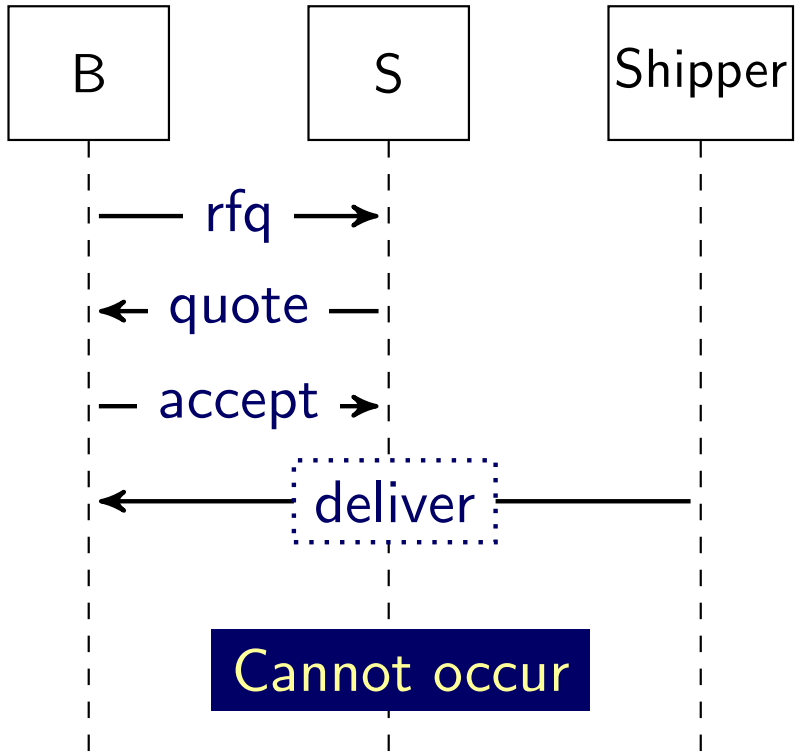
Safety and Liveness Violations

Encode a protocol's causal structure in temporal logic and evaluate propertes

Purchase Unsafe



Purchase No Ship



Safety Violation

Liveness Violation

Encode Causal Structure as Temporal Constraints

- *Reception*. If a message is received, it was previously sent.
- *Information transmission* (sender's view)
  - Any ⌜in⌝ parameter occurs prior to the message
  - Any ⌜out⌝ parameter occurs simultaneously with the message
- *Information reception* (receiver's view)
  - Any ⌜out⌝ or ⌜in⌝ parameter occurs before or simultaneously with the message
- *Information minimality*. If a role observes a parameter, it must be simultaneously with *some* message sent or received
- *Ordering*. If a role sends any two messages, it observes them in some order

Verifying Safety

- Competing messages: those that have the same parameter as out
- *Conflict*. At least two competing messages occur
- *Safety* iff the causal structure $\wedge$ conflict is unsatisfiable

Verifying Liveness

- *Maximality*. If a role is enabled to send a message, it sends at least one such message
- *Reliability*. Any message that is sent is received
- *Incompleteness*. Some public parameter fails to be bound
- *Live* iff the causal structure $\wedge$ the above three is unsatisfiable

Exercises 1: *Abruptly Cancel*

```
Abruptly Cancel {
 role B, S
 parameter out ID key, out item, out outcome

 B ↦ S: order [out ID, out item]
 B ↦ S: cancel [in ID, in item, out outcome]
 S ↦ B: goods [in ID, in item, out outcome]
}
```

- Is this protocol safe?
- Is this protocol live?

Exercise 2: *Abruptly Cancel* Modified (with ⌜nil⌝)

```
Abruptly Cancel {
 role B, S
 parameter out ID key, out item, out outcome

 B ↦ S: order [out ID, out item]
 B ↦ S: cancel [in ID, in item, nil outcome]
 S ↦ B: goods [in ID, in item, out outcome]
}
```

- Is this protocol safe?
- Is this protocol live?

Exercise 3: Goods Priority

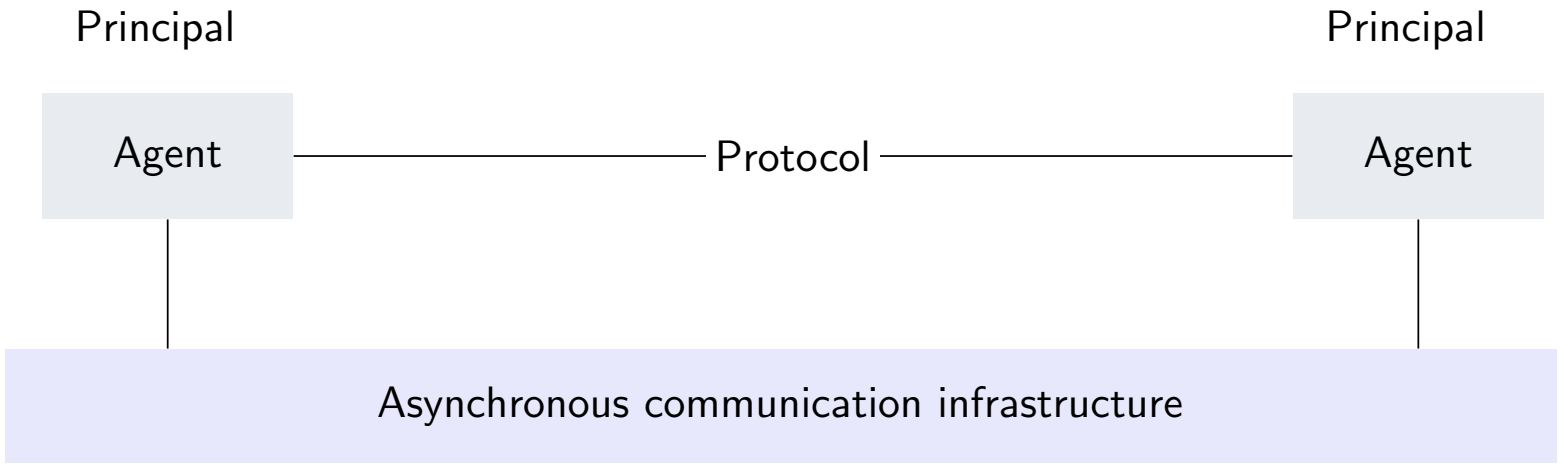
- Modify *Abruptly Cancel* so that goods takes priority over cancel
  - If S sends Goods, that is the outcome of the interaction
  - S cannot send Goods after receiving Cancel
  - If S receives Cancel before Goods, cancellation is the outcome
  - B cannot send Cancel after receiving Goods

Solution

```
Abruptly Cancel {
 role B, S
 parameter out ID key, out item, out outcome

 B ↦ S: order [out ID, out item]
 B ↦ S: cancel [in ID, in item, nil outcome, out rescind]
 S ↦ B: cancelAck [in ID, in item, out outcome, in rescind]
 S ↦ B: goods [in ID, in item, nil rescind, out outcome]
}
```

Outline

Ideal Protocol-Based System Architecture



- Constraints
  1. Agent ensures the correctness of its emissions. To do so, it needs nothing but its local state (history of prior emissions and receptions)
  2. The reception of any message is correct, if it was emitted correctly
  3. Asynchrony: Emissions nonblocking; receptions nondeterministic
     - No ordered delivery guarantee needed from infrastructure.
  4. The protocol is the complete operational specification of the system
- Assumption: Infrastructure delivers only sent messages
  - No guaranteed delivery assumed

Diverse Protocol Languages

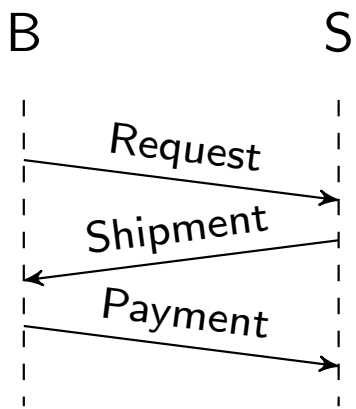
Appear in Software Engineering, Multiagent Systems, Services, Programming Languages

- Procedural, message ordering-based
  - UML interaction diagrams, MSCs
  - Trace expressions-based
    - Castagna et al., 2012
  - Session types-based (Honda et al.)
    - Scribble (Yoshida et al., 2013)
- Declarative, information-based
  - Blindingly Simple Protocol Language, or *BSPL* (Singh, 2011-12)

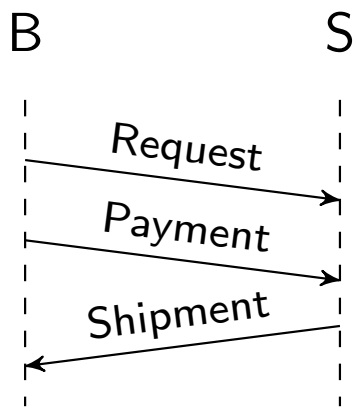
Problem: How Do Modern Languages Compare?

- Criteria
  - Concurrency
  - Asynchrony, message ordering
  - Instances (correlation and integrity)
  - Extensibility
- Based on encoding of elementary scenarios

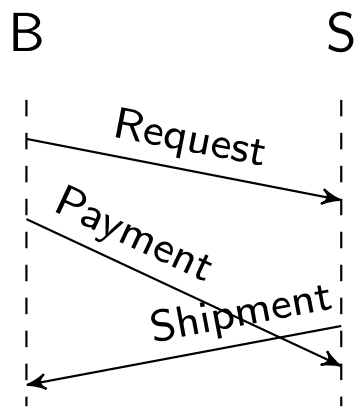
Concurrency: Can All These Enactments Be Supported?



(a) Shipment first



(b) Payment first



(c) Concurrent

Flexible Purchase: No Valid Encoding in Scribble

In Scribble, choice cannot be *mixed*, that is, between a send and a receive action

```
Flexible Purchase {
 role B, S
 parameter out ID key, out item, out shipped, out paid

 B ↦ S: Request[out ID, out item]
 S ↦ B: Shipment[in ID, in item, out shipped]
 B ↦ S: Payment[in ID, in item, out paid]
}
```

```
protocol FlexiblePurchase (role B, role S) {
  Request() from B to S;
  choice at B {
    Payment() from B to S;
    Shipment() from S to B;
  } or {
    Shipment() from S to B; // not valid
    Payment() from B to S;}}
```

Technical Reason: Possibility of deadlock

```
projection FlexiblePurchase_B {
  Request() to S;
  choice at B {   /*internal choice*/
    Payment() to S;
    Shipment() from S;
  } or {
    Shipment() from S;
    Payment() to S;
  }
}

projection FlexiblePurchase_S (role B, role S) {
  Request() from B ;
  choice at B {   /*external choice*/
    Payment() from B;
    Shipment() to B;
  } or {
    Shipment() to B;
    Payment() from B;
  }
}
```

Flexible Purchase: No Valid Encoding in Trace

For analogous reasons

$$\text{// Flexible Purchase in Trace}$$

$$\text{Buyer} \xrightarrow{\text{Request}} \text{Seller}; (\text{Buyer} \xrightarrow{\text{Payment}} \text{Seller} \wedge \text{Seller} \xrightarrow{\text{Shipment}} \text{Buyer})$$

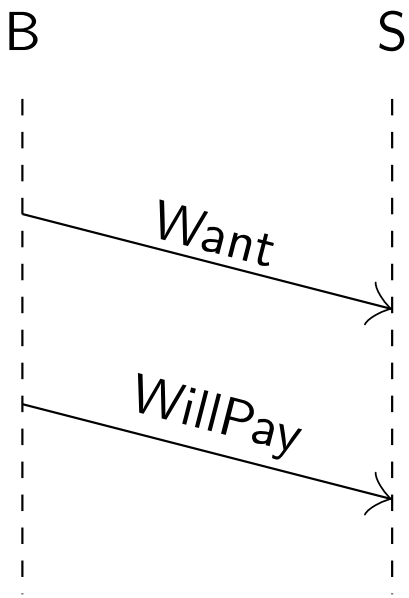
$$\text{// Projections}$$

$$\text{Buyer}: \text{Seller}!\text{Request}.((\text{Seller}?\text{Shipment}.\text{Seller}!\text{Payment}) \oplus (\text{Seller}!\text{Payment}.\text{Seller}?\text{Shipment}))$$

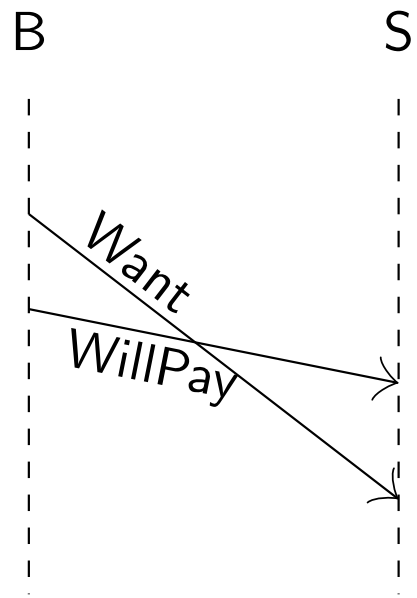
$$\text{Seller}: \text{Buyer}?\text{Request}.((\text{Buyer}!\text{Shipment}.\text{Buyer}?\text{Payment}) + (\text{Buyer}?\text{Payment}.\text{Buyer}!\text{Shipment}))$$

Scribble Needs FIFO Ordering; BSPL doesn't

B sends *Want* (some item) and then *WillPay* (some amount) to S



(d) FIFO delivery



(e) Non-FIFO delivery

Want+WillPay in BSPL and Scribble

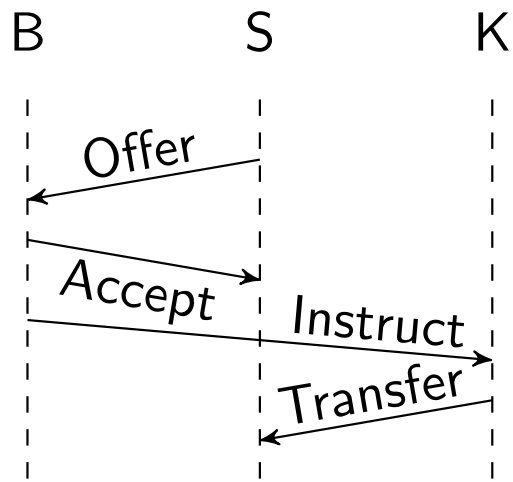
Without FIFO, Scribble violates *Reception Correctness*

```
WantWillPay {
 role B, S
 parameter out ID, out item, out price

 B ↦ S: Want[out ID, out item]
 B ↦ S: WillPay[in ID, in item, out price]
}
```

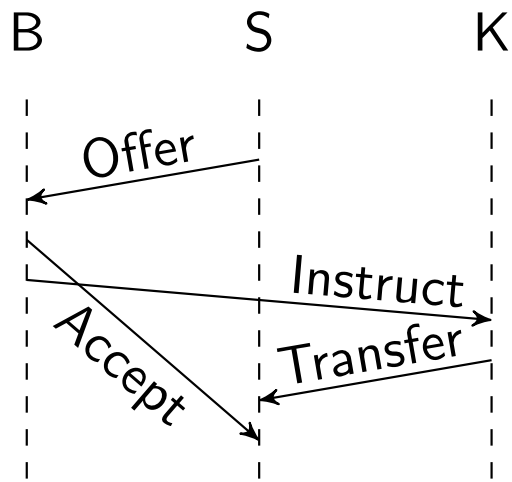
```
protocol WantWillPay (role B, role S) {
  Want() from B to S;
  WillPay() from B to S;
}

projection WantWillPay_S {
  Want() from B;
  WillPay() from B;
}
```

How Far Does Pairwise FIFO Go Though?

Scenario: In an indirect-payment purchase protocol, after receiving an *Offer* B sends *Accept* to S and then *Instruct* (a payment instruction) to bank K. Upon receiving *Instruct*, B does a funds *Transfer* to S



(f) In-order delivery



(g) Out-of-order delivery (despite satisfying FIFO)

BSPL Captures Scenario; Scribble Introduces a Contortion

Scribble: Reorders messages based upon channels

```
Indirect Payment {
 role B, S, K // K is bank
 parameter out ID key, out item, out price, out acc, out inst, out OK

 S ↦ B: Offer[out ID, out item, out price]
 B ↦ S: Accept[in ID, in item, in price, out acc]
 B ↦ K: Instruct[in ID, in price, in acc, out inst]
 K ↦ S: Transfer[in ID, in price, in inst, out OK]
}
```

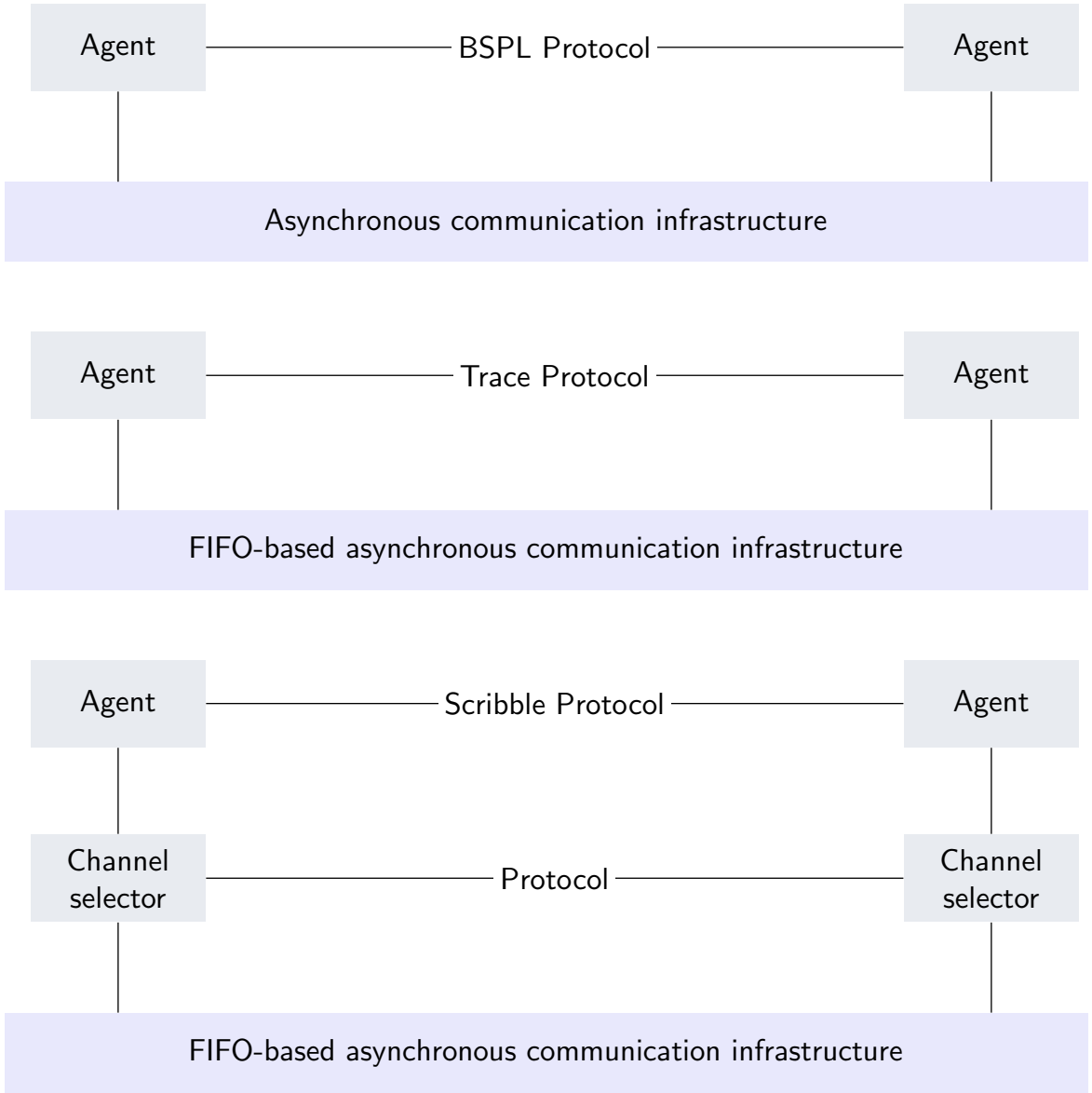
```
protocol IndirectPayment(role S, role C, role B) {
  Offer() from S to B;
  Accept() from B to S;
  Instruct() from B to K;
  Transfer() from K to S;
}
projection IndirectPayment_S(role B, role S, role K) {
  Offer() to B;
  Accept() from B;
  Transfer() from K;
}
```

Want+WillPay and Indirect Payment in Trace

Without FIFO, Trace too violates *Reception Correctness* and, slightly differently from Scribble, no possible encoding of Indirect Payment in Trace

```
//Want+WillPay protocol in Trace
Buyer --Want--> Seller ; Buyer --WillPay--> Seller

//Want+WillPay Projections
Buyer: Seller!Want ; Seller!WillPay
Seller: Buyer?Want ; Buyer?WillPay


//Indirect Payment
Seller --Offer--> Buyer; Buyer --Accept--> Seller; Buyer --Instruct--> Bank;
  Bank --Transfer--> Seller

//Indirect Payment Projection
Seller: Buyer!Offer; Buyer?Accept; Bank?Transfer
```

Architectural Comparison



Principles for Protocol Languages (1)

- *Nonunitarian*: A protocol must not specify computations from a unitary perspective
  - Scribble and Trace both give the computations of the protocol from a unitary perspective—as a sequence of messaging events.
  - Deadlock occurs in Scribble and Trace versions of *Flexible Purchase* because a computation of the protocol cannot be realized by agents reasoning locally.
- *Noninterference*: Protocol must not block legitimate agent reasoning
  - By allowing the FIFO infrastructure (and channel selector, in case of Scribble) to hold back messages from an agent blocks the reasoning the agent could have performed if messages were delivered to it as they came, Scribble and Trace violate the principle.
  - Seller could have processed *WillPay* and gone ahead with shipping even if it had not received *Want*
  - Seller could have processed *Transfer* and gone ahead with shipping even if it had not received *Accept*

Principles for Protocol Languages (2)

- *Protocol End-to-End*: Correct protocol enactment must not rely on message ordering guarantees from the communication infrastructure since the appropriate constraints are to be implemented and checked in agents.
  - As Indirect Payment illustrates, FIFO infrastructure is inadequate.
  - FIFO ordering is also excessive: orders messages that bear no relation to each other.

```
Just-Want {
 role Buyer, Seller
 parameter out ID, out item

 Buyer ↦ Seller: Want[out ID, out item]
}

Hello-World {
 role Buyer, Seller
 parameter out gID, out utterance

 Buyer ↦ Seller: Greeting[out gID, out utterance]
}
```

BSPL Less Demanding, Yet Offers More

Details: https://arxiv.org/abs/1901.08441

Criterion	Trace	Scribble	BSPL
Supports concurrency	No	No	Yes
Unordered delivery	No	No	Yes
Instances	Limited	Limited	Yes
Integrity	No	No	Yes
Norms computation	No	No	Yes
Extensibility	No	No	Yes
Nonunitarian	No	No	Yes
Noninterference	No	No	Yes
Protocol E2E	No	No	Yes

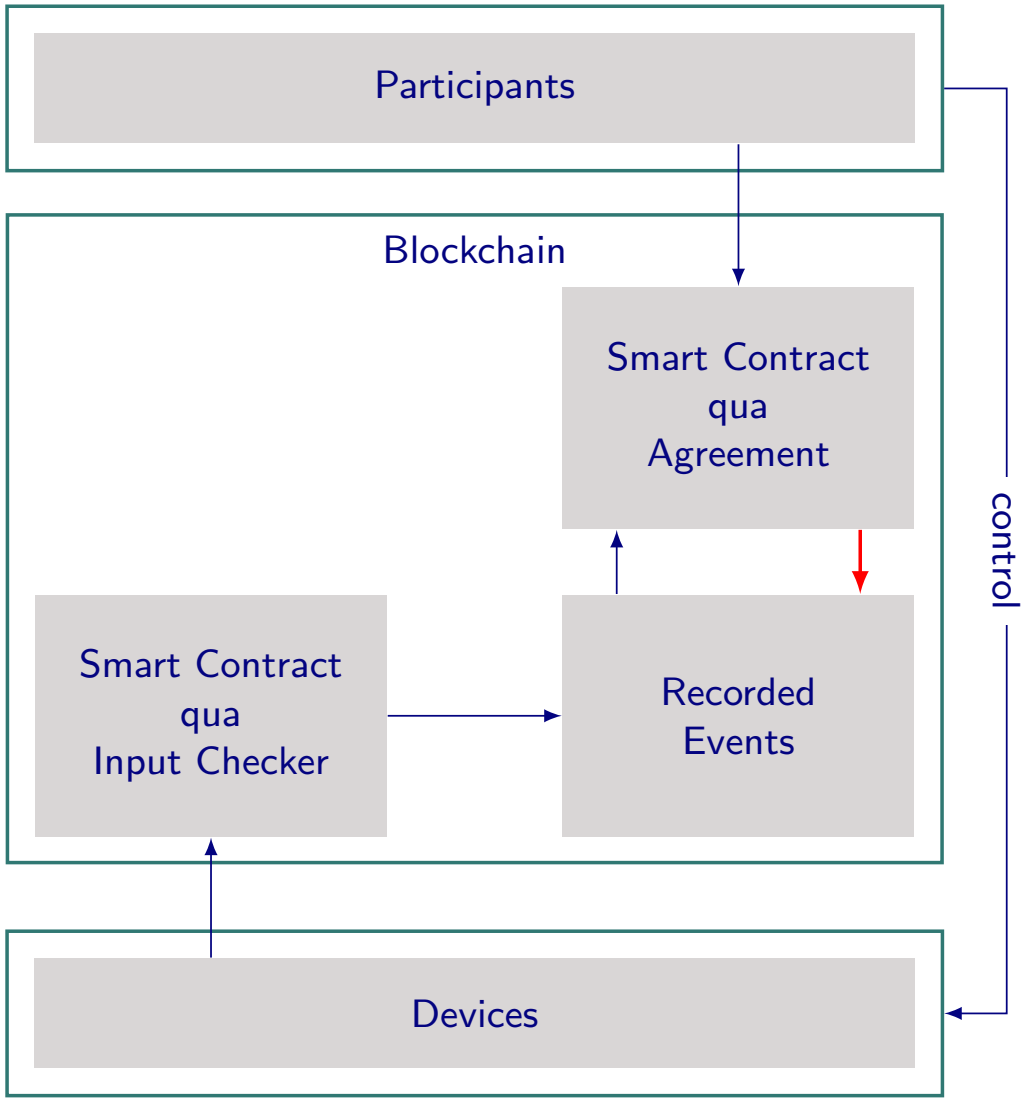
Outline

Smart Contract

Inviolability as contract

- Program signed by principals and stored on blockchain
- Known in advance how it will function
- Guaranteed to run if inputs satisfied
  - Doomsday machine?
- Bitcoin transactions
- On Ethereum, R3 Corda, etc., general-purpose platforms

Smart Contract-Based Architecture



Limitations of Smart Contracts

Consider "Goods delivered if payment made"

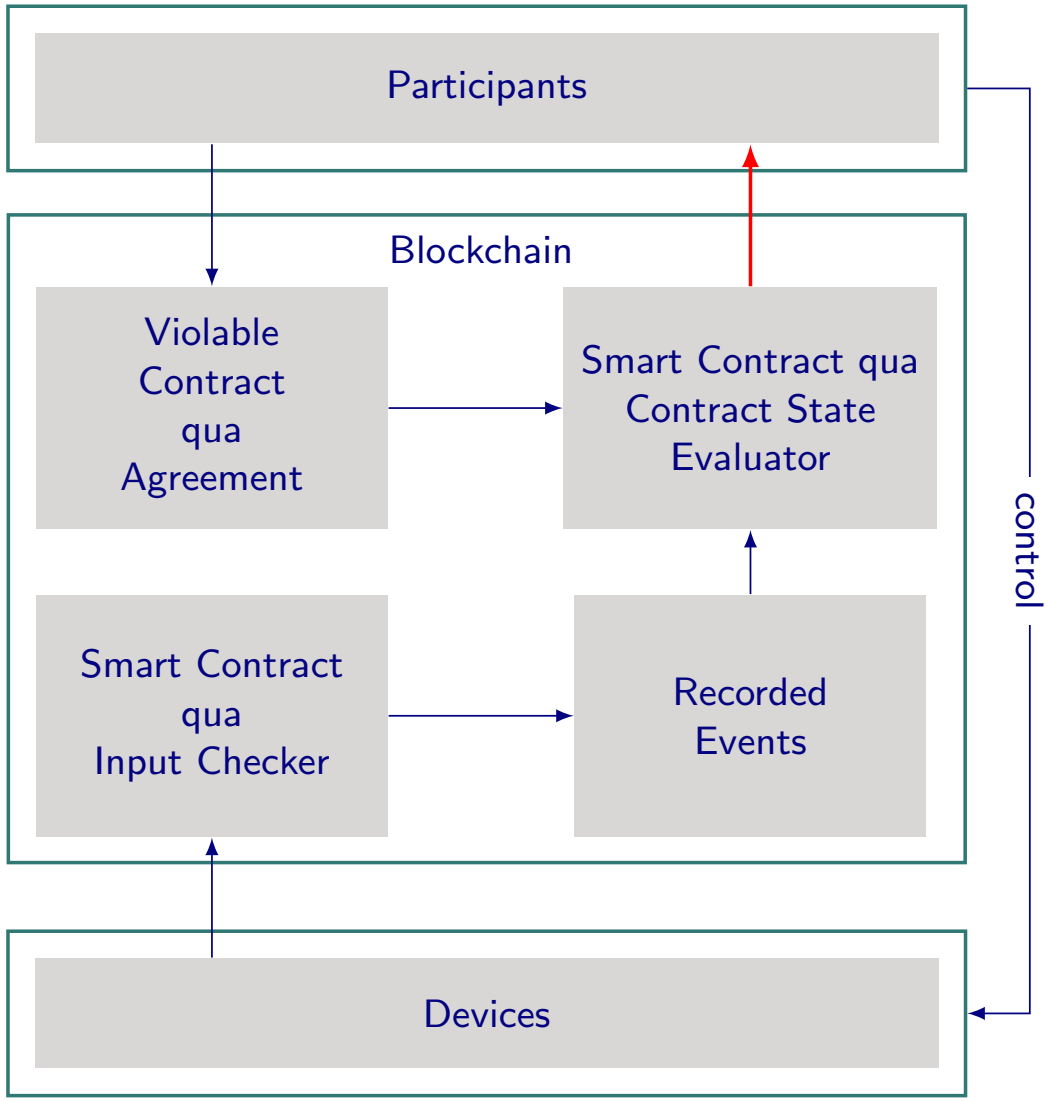
- Take away participant autonomy
  - Seller may not deliver goods
- Difficult to understand, verify, and validate
  - Low-level programs
- Lack of social meaning
  - Real world cannot be abstracted away
    - What if there are no goods in Seller's inventory?
    - What if delivery fails?
    - What if buyer denies having received goods?
  - Undoing Ethereum transactions in DAO hack was a social response
    - An ad hoc one though
    - So much for immutability!

Challenges for Contract-Driven Computing

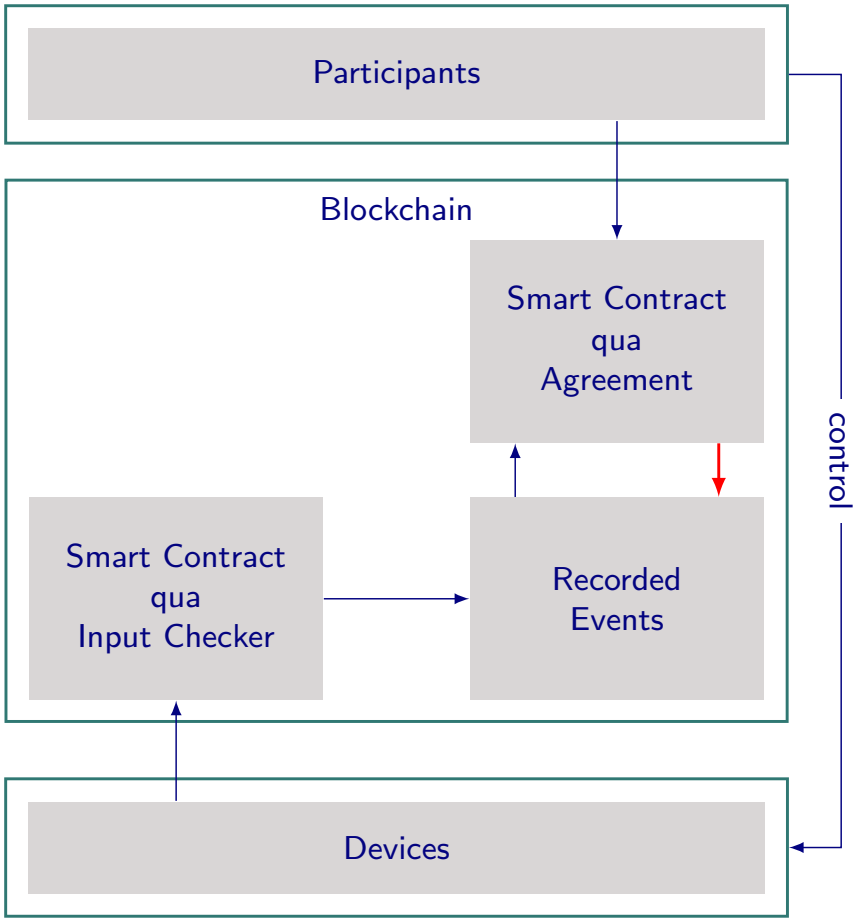
How to represent and compute the social, rather than get rid of it

- Declarative representations of contracts that
  - Accommodate autonomy: Contracts may be violated
  - Support verifiability: Violations are recorded
  - Support social meaning (expectations, trust, accountability)
  - Support stakeholder validation via high-level representations
  - Can be computed in a decentralized manner

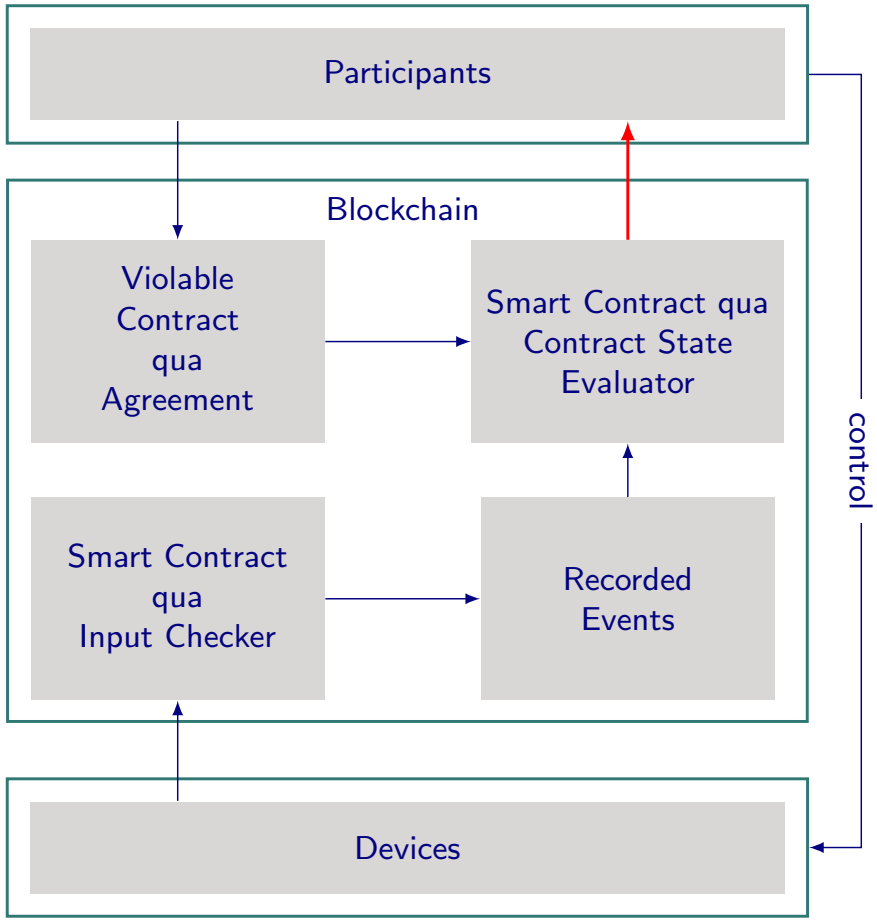
Violable Contracts on Blockchain



Smart Contracts vs. Violable Contracts, Architecturally



(h) Smart contracts



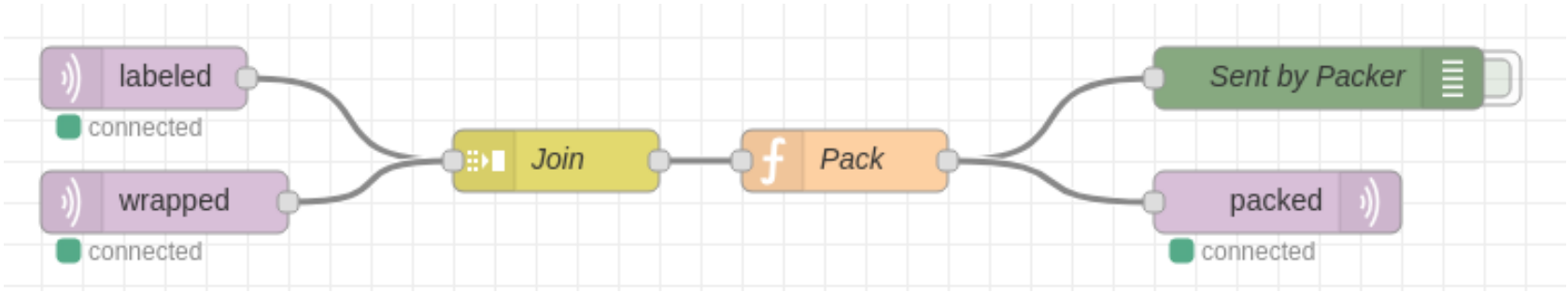
(i) Compacts

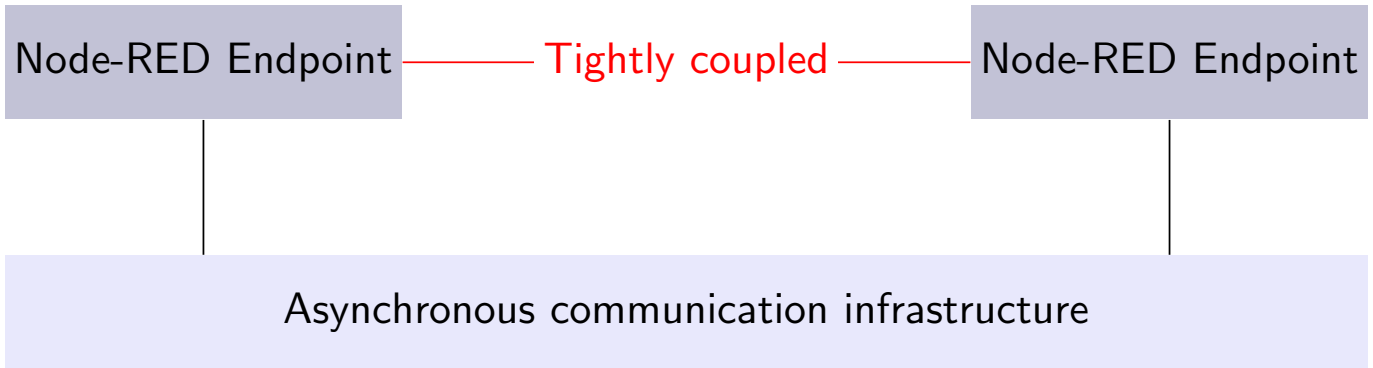
Contrasting Compacts with Traditional and Smart Contracts

	Traditional	**Smart**	**Compacts**
Specification	Text	Procedure	Formal, declarative
Automation	None	Full	Compliance checking
Principals' Control	Complete	None	Complete
Venue	External	Within blockchain	Recorded on blockchain
Trust Model	Hidden	Hardcoded	Explicit
Social Meaning	Informal	None	Formal
Correctness Standard	Informal legal	Whatever executes	Formal legal
Scope	Open but ad hoc	Closed	Sociotechnical

IoT Vision: Decentralized Applications Supported by Fine-Grained Information about the Environment

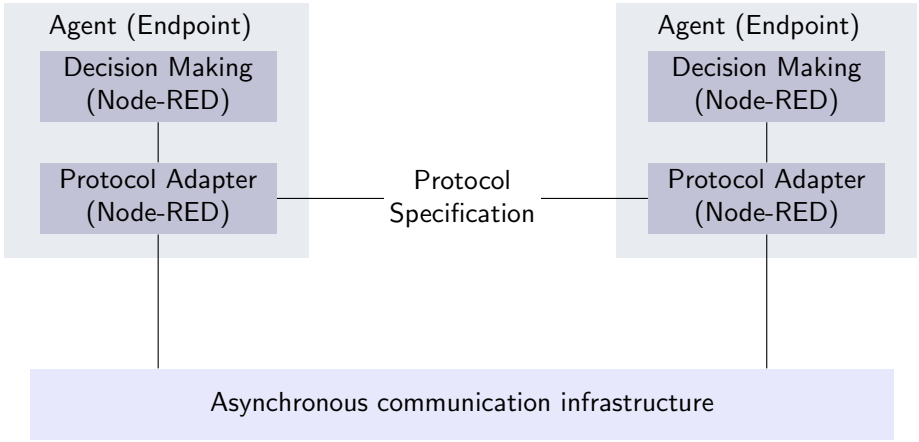
However, popular programming models such as Node-RED are based upon orchestration

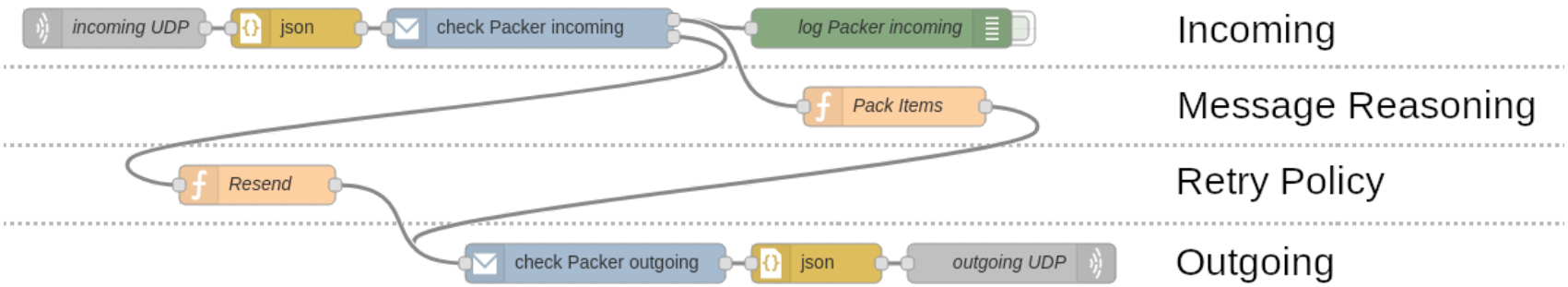




Realizing BSPL via LoST in Node-RED

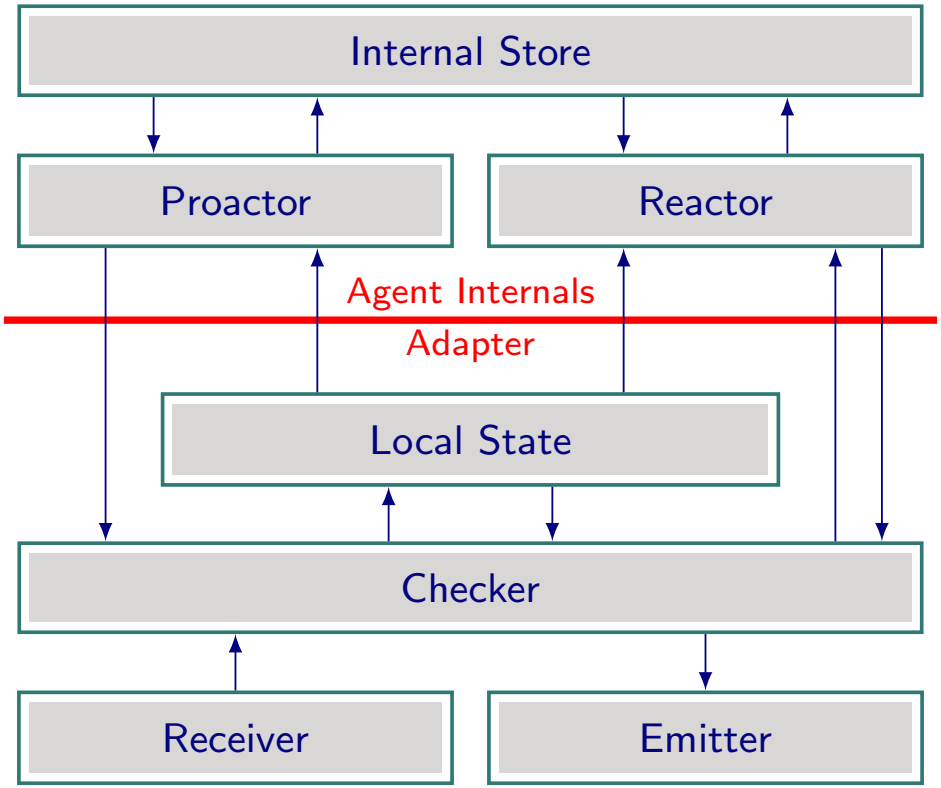
Implemented over UDP: Supports application-level retries to mitigate message loss
Performance comparable to MQTT over TCP





Decentralized Applications on FaaS Platforms

Protocol + FaaS = highly modular and concurrent agent out of the box
Developer focuses on business logic
Implemented on AWS Lambda



Outline

Takeaways: 1

- Any system that spans autonomous parties is decentralized
- A single machine (even if distributed) cannot model decentralization
- Decentralization requires modeling interactions
- Norms capture meaning of interaction
- Norms must be represented
  - Crucial to modeling agreements
  - Support compliance checking, trust, and accountability
- Norms have be operationalized over protocols

Takeaways: 2

- Protocols must not rely on ordering guarantees from infrastructure
  - Don't hide synchronization in the infrastructure
  - Can the protocol work over UDP?
- Protocol must specify information causality, not a control flow of messages
- Protocol specifies how to compute decentralized objects
- Must a reception happen in a certain order relative to other receptions and emissions for purposes of correctness? Then you are yet to embrace asynchrony
- Don't impose models that interfere with agent autonomy
- Apply End-to-End Principle: Retransmissions, fault tolerance, etc. not to be implemented in infrastructure. Application-level protocols must accommodate these ideas

Exercise: Collective Concept Map

- What theme do you remember most from today?
- What additional high-level themes should we consider within
  - Software engineering?
  - Programming languages?
  - Artificial intelligence?
  - Distributed computing?
- What research questions are worth pursuing with the aim of promoting a deeper understanding between Interaction Orientation and PL?

Thanks!

- Samuel Christie, Daria Smirnova
- National Science Foundation
- EPSRC
- Science of Security Lablet
- Consortium for Ocean Leadership
- Raymond Hu, Viviana Mascardi, and Angelo Ferrando for feedback on protocol languages evaluation