Interactions and Protocols

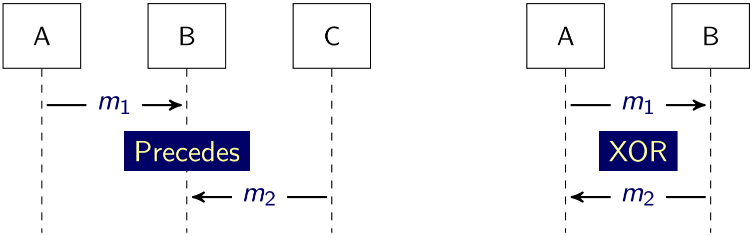
Distributed systems of autonomous, heterogeneous agents

- Enable interoperation
- Maintain independence from internal reasoning (policies)
- Support composition of distributed systems

Consider protocols as constructed over messages

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 string valued)

- ⌜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

The *Hello* Protocol

```
Hello {
 role Self, Other
 parameter out greeting key

 Self ↦ Other: hi[out greeting key]
}
```

- At most one instance of *Hello* for each greeting
- At most one *hi* message for each greeting
- Enactable standalone: no parameter is ⌜in⌝
- The key of *hi* is explicit; often left implicit on messages

The *Pay* Protocol

```
Pay {
 role Payer, Payee
 parameter in ID key, in amount

 Payer ↦ Payee: payM[in ID, in amount]
}
```

- At most one *payM* for each ID
- Not enactable standalone: **why?**
- The key of *payM* is implicit (for brevity)

The *Offer* Protocol

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

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

- The key ID uniquifies instances of *Initiate Offer*, *rfq*, and *quote*
- Not enactable standalone: at least one parameter is ⌜in⌝
- An instance of *rfq* must precede any instance of *quote* with the same ID: **why?**
- No message need occur: **why?**
- *quote* must occur for *Offer* to complete: **why?**

The *Initiate Order* Protocol

```
Initiate-Order {
 role B, S
 parameter out ID key, out item, out price, out rID

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

- The key ID uniquifies instances of *Order* and each of its messages
- Enactable standalone
- An *rfq* must precede a *quote* with the same ID
- A *quote* must precede an *accept* with the same ID
- A *quote* must precede a *reject* with the same ID
- An *accept* and a *reject* with the same ID *cannot* both occur: **why?**

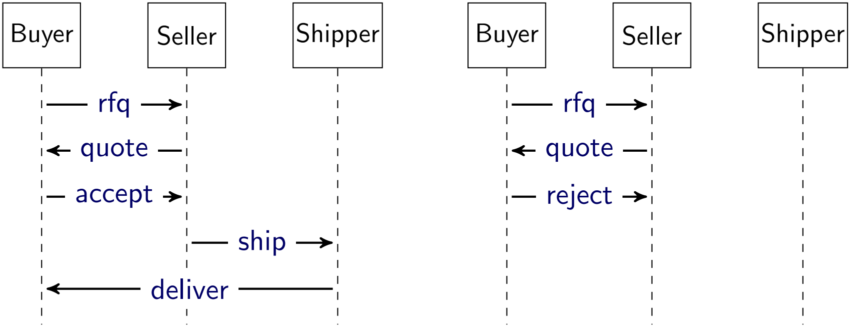
The *Purchase* Protocol

```
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]
}
```

- At most one item, price, and outcome binding per ID
- Enactable standalone
- *reject* conflicts with *accept* on response (a *private* parameter)
- *reject* or *deliver* must occur for completion (to bind outcome)

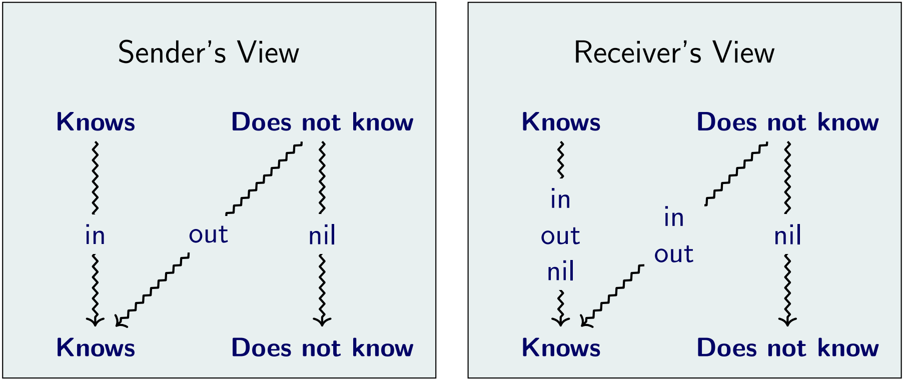
Possible Enactments as Sets of Local Histories

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



Knowledge and Viability

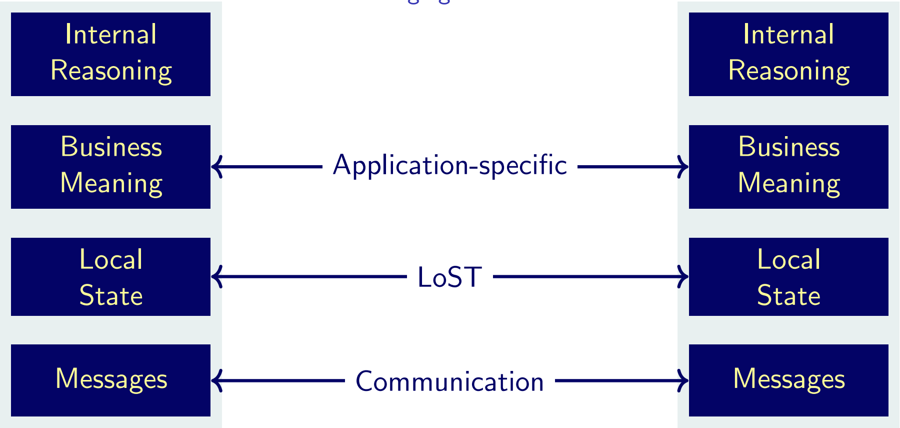
When is a message viable? What effect does it have on a role's local knowledge?



- 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

Realizing BSPL via LoST (Local State Transfer)

Does *not* assume FIFO or reliable messaging



- Unique messages
- Integrity checks on incoming messages
- Consistency of local choices on outgoing messages

Implementing 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

Benefits

- Technical
  - Statelessness
  - Consistency
  - Atomicity
  - Natural composition
- Conceptual
  - Make protocol designer responsible for specifying causality
  - Avoid contortions of traditional approaches when causality is unclear

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

Send-Receive and Send-Send Constraints on a Role

Considering two or more schemas with the same parameter

	Sends in	*Sends* out	*Sends* nil
Sends in	Unconstrained	Send out first	Send nil first
Sends out		Send at most one	Send nil first
Sends nil			Unconstrained
Receives in	Receive at least one instance before send	Receive may occur after send	Send before receive
Receives out	Receive at least one instance before send	Impossible	Send before receive
Receives nil	Unconstrained	Unconstrained	Unconstrained

Summarizing Approaches for Interaction

	Declarative (Explicit)	*Procedural* (Implicit)
Meaning	Commitments and other norms	Hard coded within internal reasoning heuristics
Operation	Temporal logic BSPL	State machines; Petri nets; process algebras

- Declarative approaches for meaning
  - Improve flexibility
  - Under-specify enactment: potential of interoperability failures
- Procedural or declarative approaches for operations
  - Operationally clear, but
    - Tend to emphasize control flow
    - Tend to over-specify operational constraints
    - Yield nontrivial interoperability and endpoint projections

Well-Formedness Conditions

- A parameter that is adorned ⌜in⌝ in a declaration must be ⌜in⌝ throughout its body
- A parameter that is adorned ⌜out⌝ in a declaration must be ⌜out⌝ in at least one reference
  - When adorned ⌜out⌝ in zero references, not enactable
  - When adorned ⌜out⌝ in exactly one reference, consistency is guaranteed
  - When adorned ⌜out⌝ in two or more references, no more than one can execute
- A private parameter must be ⌜out⌝ in at least one reference and ⌜in⌝ in at least one reference

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

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
  - Must have bindings for some parameters
  - Analogous to NOT NULL constraints
- 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 Minus 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

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

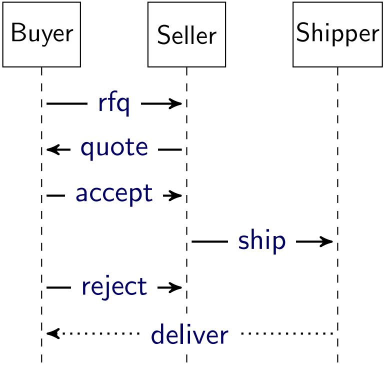
Verifying Liveness

- *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 occurrence is unsatisfiable

Safety and Liveness Violations

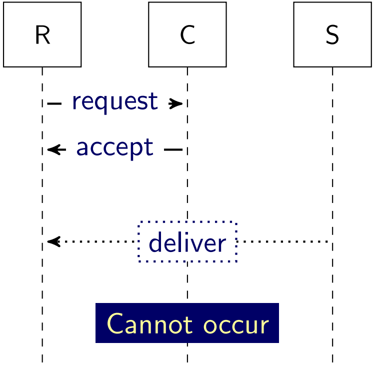
Encode a protocol's causal structure in temporal logic and evaluate properties

Purchase Unsafe



Safety Violation

Purchase Minus Ship



Liveness Violation

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

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

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

Comparing LoST and ReST

	ReST	*LoST*
Modality	Two-party; client-server; synchronous	Multiparty interactions; peer-to-peer; asynchronous
Computation	Server computes definitive resource state	Each party computes its definitive local state and the parties collaboratively and (potentially implicitly) compute the definitive interaction state
State	Server maintains no client state	Each party maintains its local state and, implicitly, the relevant components of the states of other parties from which there is a chain of messages to this party

Comparing LoST and ReST

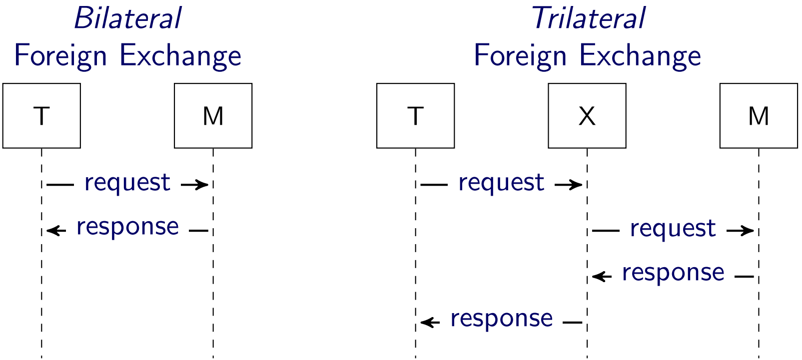
	ReST	*LoST*
Transfer	State of a resource, suitably represented	Local state of an interaction via parameter bindings, suitably represented
Idempotent	For some verbs, especially GET	Always; repetitions are guaranteed harmless
Caching	Programmer can specify if cacheable	Always cacheable
Uniform interface	GET, POST, ...	⌜in⌝, ⌜out⌝, ⌜nil⌝
Naming	Of resources via URIs	Of interactions via (composite) keys, whose bindings could be URIs

Comparing LoST and WS-CDL

- Similarity: both emphasize interaction
- Differences: WS-CDL is
  - Procedural
    - Explicit constructs for ordering
    - Sequential message ordering by default
  - Agent-oriented
    - Includes agents' internal reasoning within choreography (specify what service an agent executes upon receiving a message)
    - Relies on agents' internal decision-making to achieve composition (take a value from Chor A and send it in Chor B)
  - No semantic notion of completeness

Composing Protocols

Without imposing private constraints on a party playing a role



- Is *Trilateral* $=$ *Bilateral* $\otimes$ *Bilateral*?

The *Bilateral Price Discovery* protocol

```
Bilateral {
 role Taker, Maker
 parameter out reqID key, out query, out result

 Taker ↦ Maker: priceRequest[out reqID, out query]
 Maker ↦ Taker: priceResponse[in reqID, in query, out result]
}
```

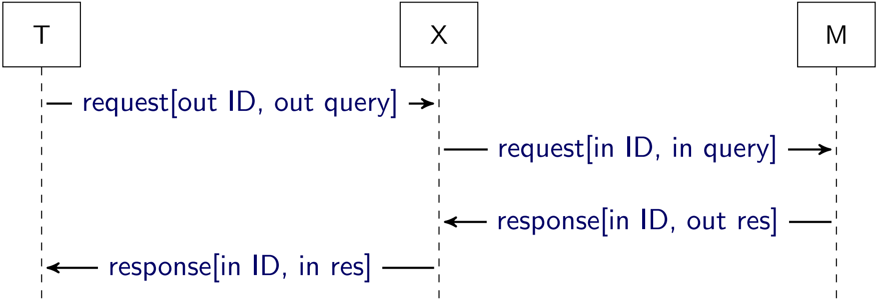
The *General Bilateral Price Discovery* protocol

```
GeneralBilateral {
 role T, M
 parameter reqID key, query, res

 T ↦ M: priceRequest[out reqID, out query]
 T ↦ M: priceRequest[in reqID, in query]

 M ↦ T: priceResponse[in reqID, in query, out res]
 M ↦ T: priceResponse[in reqID, in query, in res]
}
```

The *Trilateral* Protocol

Also called price discovery

```
Trilateral {
 role Taker, Exchange, Maker
 parameter out ID key, out query, out res

  GeneralBilateral(Taker, Exchange, out ID, out query, in res)
  GeneralBilateral(Exchange, Maker, in ID, in query, out res)
}
```



Standing Order

As in insurance claims processing

```
Insurance−Claims {
 role Vendor, Subscriber
 parameter out policyNO key, out reqForClaim key, out claimResponse

 Vendor ↦ Subscriber: createPolicy[out policyNO]
 Subscriber ↦ Vendor: serviceReq[in policyNO, out reqForClaim]
 Vendor ↦ Subscriber: claimService[in policyNO, in reqForClaim, out
      claimResponse]
}
```

- Each claim corresponds to a unique policy and has a unique response
- One policy may have multiple claims
- Could make {policyNO, reqForClaim} both key

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

Shopping Cart

```
Shopping Cart {
 role B, S
 parameter out ID key, out lineID key, out item, out qty, out price, out
     finalize

 B ↦ S: create[out ID]
 S ↦ B: quote[in ID, out lineID, in item, out price]
 B ↦ S: add[in ID, in lineID, in item, out qty, in price]
 B ↦ S: remove[in ID, in lineID]

 S ↦ B: total[in ID, out sum]
 B ↦ S: settle[in ID, in sum, out finalize]
 B ↦ S: discard[in ID, out finalize]
}
```

Exercise 1: *Abruptly Cancel*

Solution added

```
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? **No**
- Is this protocol live? **Yes**

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

Solution added

```
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? **Yes**
- Is this protocol live? **Yes**
  - But it lacks business realism because the SELLER may send *goods* even after receiving *cancel*

The *Bid Offer* protocol

```
Bid Offer {
 role Coordinator uni, Bidder ⊒ Winner uni
 parameter out ID key, out request, out response, out decision

 Coordinator ↦ Bidder: CfB[out ID, out request]

 Bidder ↦ Coordinator: bid[in ID, in request, out response]

 Coordinator ↦ Winner: offer[in ID, in request, in response, out
     decision]
}
```