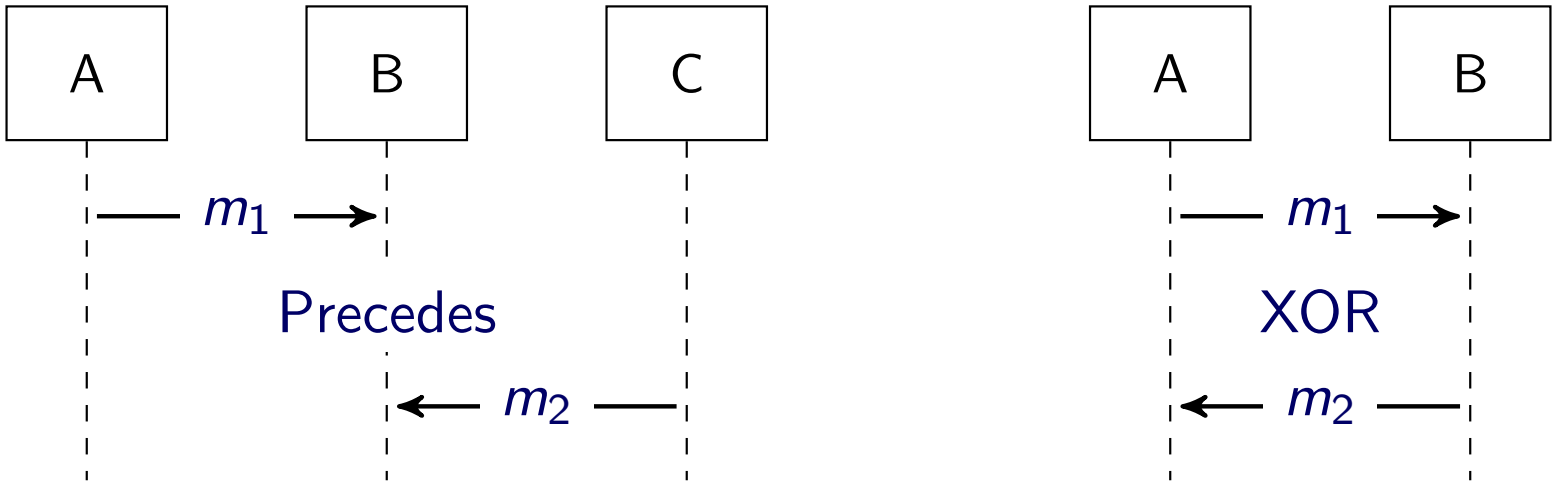
Traditional Specifications: Procedural

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 {
roles Self, Other
parameters 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 {
roles Payer, Payee
parameters 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 {
roles Buyer, Seller
parameters 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 {
roles B, S
parameters 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?**

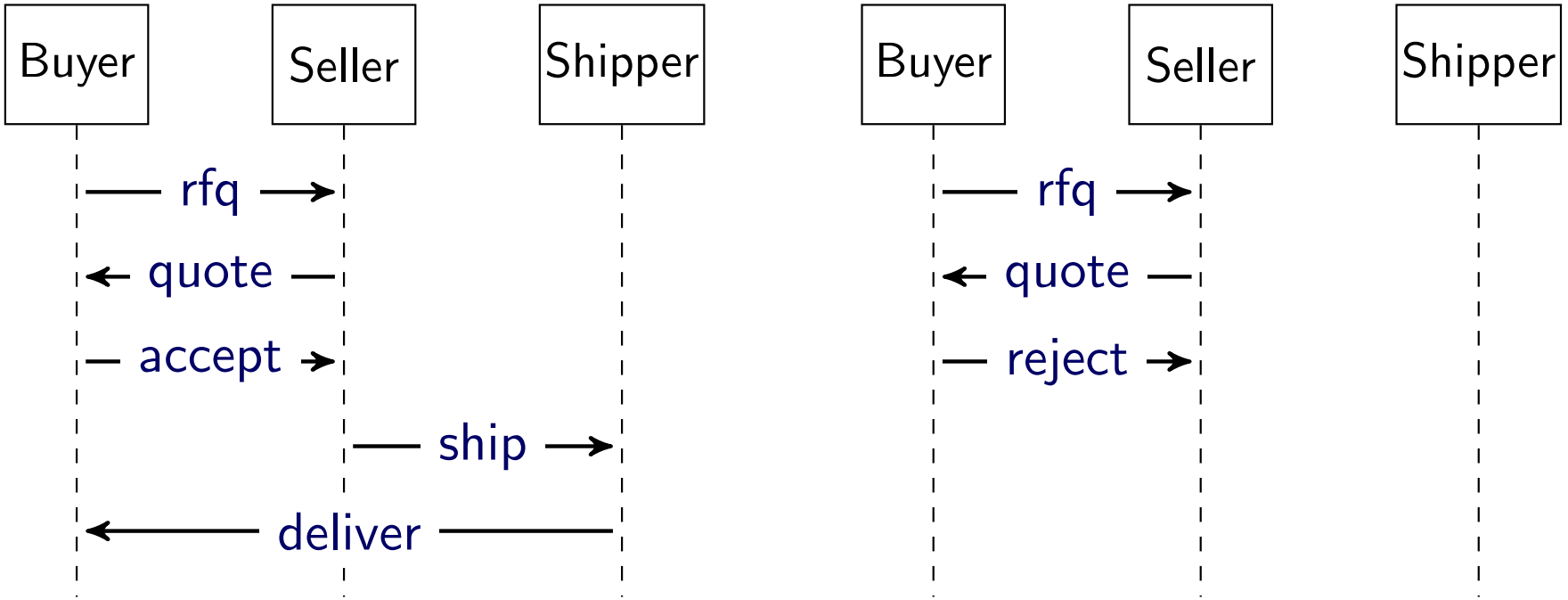
The *Purchase* Protocol

```
Purchase {
roles B, S, Shipper
parameters 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

Possible Enactments as Sets of Local Histories

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



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

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