LoST: Local State Transfer
And BSPL, the Blindingly Simple Protocol Language

Munindar P. Singh

North Carolina State University

Spring 2014
Interactions and Protocols

Distributed systems of autonomous, heterogeneous agents

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

- Autonomy
- Myopia
  - All choices must be local
  - Correctness should not rely on future interactions
- Heterogeneity: local $\neq$ internal
  - Local state (projection of global, 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
Traditional Specifications

Low-level, procedural approaches leading to over-specified protocols

- 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
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

- ▶ \textit{in\textsuperscript{\textbullet}}: 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

- ▶ \textit{out\textsuperscript{\textbullet}}: Information that is generated by the protocol instances
  - Bindings can be fed into other protocols through their \textit{in\textsuperscript{\textbullet}} parameters, thereby accomplishing composition
  - A standalone protocol must adorn all its public parameters \textit{out\textsuperscript{\textbullet}}

- ▶ \textit{nil\textsuperscript{\textbullet}}: Information that is absent from the protocol instance
  - Bindings must not exist

Ignoring data types of parameters for simplicity: assume strings everywhere
The **Hello** Protocol

Hello \{ 
\begin{align*}
\text{role} & \text{ Self, Other} \\
\text{parameter} & \text{ out greeting key}
\end{align*}
\}

Self $\mapsto$ Other: hi[\text{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 $\text{in}$
- The key of *hi* is explicit; often left implicit on messages
The Pay Protocol

Pay \{ 
  \textit{role} \ Payer \ , \ Payee \\
  \textit{parameter} \ in \ ID \ \textit{key}, \ in \ amount \\

  Payer \mapsto \ Payee: \text{payM}[in \ ID, \ in \ amount] \\
\}

- At most one \textit{payM} for each ID
- Not enactable standalone: \textbf{why}?
- The key of \textit{payM} is implicit; could be made explicit
- Eliding \texttt{⌜means⌝} clauses in this paper
The *Offer* Protocol

\[
\text{Offer } \{ \\
\text{ role Buyer, Seller} \\
\text{ parameter in ID key, out item, out price } \\
\}
\]

- Buyer \(\mapsto\) Seller: rfq [in ID, out item]
- Seller \(\mapsto\) Buyer: quote [in ID, in item, out price]

- The key ID unqiues 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

\[
\text{Initiate Order } \{ \\
\text{role B, S} \\
\text{parameter out ID key, out item, out price, out rID} \\
\]

\[
\begin{align*}
B & \rightarrow S: \text{rfq [out ID, out item]} \\
S & \rightarrow B: \text{quote [in ID, in item, out price]} \\
B & \rightarrow S: \text{accept [in ID, in item, in price, out rID]} \\
B & \rightarrow S: \text{reject [in ID, in item, in price, out rID]}
\end{align*}
\]

- The key ID uniques 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

\[ \text{Purchase} \{ \]

\[ \text{role} \ B, \ S, \ \text{Shipper} \]

\[ \text{parameter} \ \text{out} \ ID, \ \text{key}, \ \text{out} \ \text{item}, \ \text{out} \ \text{price}, \ \text{out} \ \text{outcome} \]

\[ \text{private} \ \text{address}, \ \text{resp} \]

\[ B \rightarrow S: \text{rfq} [\text{out} \ ID, \ \text{out} \ \text{item}] \]

\[ S \rightarrow B: \text{quote} [\text{in} \ ID, \ \text{in} \ \text{item}, \ \text{out} \ \text{price}] \]

\[ B \rightarrow S: \text{accept} [\text{in} \ ID, \ \text{in} \ \text{item}, \ \text{in} \ \text{price}, \ \text{out} \ \text{address}, \ \text{out} \ \text{resp}] \]

\[ B \rightarrow S: \text{reject} [\text{in} \ ID, \ \text{in} \ \text{item}, \ \text{in} \ \text{price}, \ \text{out} \ \text{outcome}, \ \text{out} \ \text{resp}] \]

\[ S \rightarrow \text{Shipper}: \text{ship} [\text{in} \ ID, \ \text{in} \ \text{item}, \ \text{in} \ \text{address}] \]

\[ \text{Shipper} \leftrightarrow B: \text{deliver} [\text{in} \ ID, \ \text{in} \ \text{item}, \ \text{in} \ \text{address}, \ \text{out} \ \text{outcome}] \]

\[ \}

- 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 Enactment as a Vector of Local Histories

Buyer

$rfq$

$\downarrow$

ID, item

$\downarrow$

quote

ID, price

$\downarrow$

accept

ID, address

$\downarrow$

ship

ID, item, address

$\downarrow$

deliver

ID, item, address, outcome

Seller

$rfq$

$\downarrow$

ID, item

$\downarrow$

quote

ID, price

$\downarrow$

accept

ID, address

$\downarrow$

ship

ID, item, address

$\downarrow$

deliver

Shipper
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 \texttt{out} parameter \textbf{creates} and transmits knowledge.
- An \texttt{in} parameter transmits knowledge.
- Repetitions through multiple paths are harmless and superfluous.
Realizing BSPL via LoST

LoST = Local State Transfer

- Does not assume FIFO or reliable messaging
- Provides
  - 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
- Send any unique message provided
  - Parameter bindings agree with previous bindings for the same keys for \(\text{in}\) parameters
  - No bindings for \(\text{out}\) and \(\text{nil}\) parameters exist
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**
## Comparing LoST and ReST

<table>
<thead>
<tr>
<th>Modality</th>
<th>ReST</th>
<th>LoST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Two-party; client-server; synchronous</td>
<td>Multiparty interactions; peer-to-peer; asynchronous</td>
</tr>
<tr>
<td>Computation</td>
<td>Server computes definitive resource state</td>
<td>Each party computes its definitive local state and the parties collaboratively and (potentially implicitly) compute the definitive interaction state</td>
</tr>
<tr>
<td>State</td>
<td>Server maintains no client state</td>
<td>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</td>
</tr>
</tbody>
</table>
Comparing LoST and ReST

<table>
<thead>
<tr>
<th></th>
<th>ReST</th>
<th>LoST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transfer</strong></td>
<td>State of a resource, suitably represented</td>
<td>Local state of an interaction via parameter bindings, suitably represented</td>
</tr>
<tr>
<td><strong>Idempotent</strong></td>
<td>For some verbs, especially GET</td>
<td>Always; repetitions are guaranteed harmless</td>
</tr>
<tr>
<td><strong>Caching</strong></td>
<td>Programmer can specify if cacheable</td>
<td>Always cacheable</td>
</tr>
<tr>
<td><strong>Uniform interface</strong></td>
<td>GET, POST, ...</td>
<td>「in」, 「out」, 「nil」</td>
</tr>
<tr>
<td><strong>Naming</strong></td>
<td>Of resources via URIs</td>
<td>Of interactions via (composite) keys, whose bindings could be URIs</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Send(s)</th>
<th>Receives</th>
<th>Sends</th>
<th>Sends</th>
<th>Send(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>in</td>
<td>in</td>
<td>out</td>
<td>nil</td>
</tr>
<tr>
<td>Unconstrained</td>
<td>Send out first</td>
<td>Send at most one</td>
<td>Send nil first</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>Unconstrained</td>
</tr>
<tr>
<td>Receive at least one instance before send</td>
<td>Receive may occur after send</td>
<td>Send before receive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>out</td>
<td>in</td>
<td>out</td>
<td>nil</td>
<td>Send before receive</td>
</tr>
<tr>
<td>Receive at least one instance before send</td>
<td>Impossible</td>
<td>Unconstrained</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>Unconstrained</td>
</tr>
<tr>
<td>Unconstrained</td>
<td>Unconstrained</td>
<td>Unconstrained</td>
<td>Unconstrained</td>
<td></td>
</tr>
</tbody>
</table>
Summarizing Approaches for Interaction

<table>
<thead>
<tr>
<th></th>
<th>Declarative (Explicit)</th>
<th>Procedural (Implicit)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Meaning</strong></td>
<td>Commitments and other norms</td>
<td>Hard coded within internal reasoning heuristics</td>
</tr>
<tr>
<td><strong>Operation</strong></td>
<td>Temporal logic BSPL</td>
<td>State machines; Petri nets; process algebras</td>
</tr>
</tbody>
</table>

- 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 \( \textit{in} \) in a declaration must be \( \textit{in} \) throughout its body.
- A parameter that is adorned \( \textit{out} \) in a declaration must be \( \textit{out} \) in at least one reference.
  - When adorned \( \textit{out} \) in zero references, not enactable.
  - When adorned \( \textit{out} \) in exactly one reference, consistency is guaranteed.
  - When adorned \( \textit{out} \) in two or more references, no more than one can execute.
- A private parameter must be \( \textit{out} \) in at least one reference and \( \textit{in} \) in at least one reference.
New Contributions
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 ≠ 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
Advanced Topics
Safety: *Purchase Unsafe*

Remove conflict between *accept* and *reject*

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

  \begin{align*}
  B \rightarrow S: & \quad \text{rfq}[\textit{out} ID, \textit{out} item] \\
  S \rightarrow B: & \quad \text{quote}[\textit{in} ID, \textit{in} item, \textit{out} price] \\
  B \rightarrow S: & \quad \text{accept}[\textit{in} ID, \textit{in} item, \textit{in} price, \textit{out} address] \\
  B \rightarrow S: & \quad \text{reject}[\textit{in} ID, \textit{in} item, \textit{in} price, \textit{out} outcome] \\
  S \rightarrow \text{Shipper}: & \quad \text{ship}[\textit{in} ID, \textit{in} item, \textit{in} address] \\
  \text{Shipper} \rightarrow B: & \quad \text{deliver}[\textit{in} ID, \textit{in} item, \textit{in} address, \textit{out} outcome]
  \end{align*}

  ▶ 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
Encode Causal Structure as Temporal Constraints

- **Reception.** If a message is received, it was previously sent.
- **Information transmission** (sender’s view)
  - Any \textit{in} parameter occurs prior to the message
  - Any \textit{out} parameter occurs simultaneously with the message
- **Information reception** (receiver’s view)
  - Any \textit{out} or \textit{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 \textit{out}
- \textit{Conflict}. At least two competing messages occur
- \textit{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 $\land$ the occurrence is unsatisfiable.
Current and Future Directions

- Methodology for specifying practical protocols
- Expansion of the language to handle role hierarchies
- Treatment of recursive protocols
- For more information
  - BSPL: AAMAS 2011
  - LoST: ICWS 2011
  - Semantics: AAMAS 2012
in-out Polymorphism

price could be \texttt{[in]} or \texttt{[out]}

\textbf{Flexible-Offer} \{ 
\begin{align*}
\text{role} & \quad B, S \\
\text{parameter} & \quad \text{in ID key, out item, price, out qID} \\
B & \quad \mapsto S: \text{rfq}[ID, \text{out item, nil price}] \\
B & \quad \mapsto S: \text{rfq}[ID, \text{out item, in price}] \\
S & \quad \mapsto B: \text{quote}[ID, \text{in item, out price, out qID}] \\
S & \quad \mapsto B: \text{quote}[ID, \text{in item, in price, out qID}] \\
\end{align*}
\}

- The price can be adorned \texttt{[in]} or \texttt{[out]} in a reference to this protocol
The *Bilateral Price Discovery* protocol

BPD \{ 
    role Taker, Maker 
    parameter out reqID, key, out query, out result 

    Taker \mapsto Maker: priceRequest[\text{out reqID}, \text{out query}] 
    Maker \mapsto Taker: priceResponse[\text{in reqID}, \text{in query}, \text{out result}] 
\}
The *Generalized Bilateral Price Discovery* protocol

\[
\text{GBPDP } \begin{array}{l}
\text{role } T, M \\
\text{parameter reqID, key, query, res}
\end{array}
\]

\[
T \mapsto M: \text{priceRequest}[\text{out reqID}, \text{out query}] \\
T \mapsto M: \text{priceRequest}[\text{in reqID}, \text{in query}]
\]

\[
M \mapsto T: \text{priceResponse}[\text{in reqID}, \text{in query}, \text{out res}] \\
M \mapsto T: \text{priceResponse}[\text{in reqID}, \text{in query}, \text{in res}]
\]
The *Multilateral Price Discovery* protocol

```plaintext
MPD {
    role Taker, Exchange, Maker
    parameter out reqID key, out query, out res

    GBPD(Taker, Exchange, out reqID, out query, in res)
    GBPD(Exchange, Maker, in reqID, in query, out res)
}
```
Standing Order

As in insurance claims processing

Insurance–Claims \{ 
  \textbf{role} \ Vendor, \ Subscriber  
  \textbf{parameter} \ out \ policyNO \ key, \ out \ reqForClaim \ key, \ out \ claimResponse  
  
  Vendor \ \mapsto \ Subscriber: \ createPolicy[\ out \ policyNO]  
  Subscriber \ \mapsto \ Vendor: \ serviceReq[\ in \ policyNO, \ out \ reqForClaim]  
  Vendor \ \mapsto \ 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 \textit{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 \mapsto Seller: rfq[ in ID, out item, \textit{nil} price ]
  Buyer \mapsto Seller: rfq[ in ID, out item, out price ]

  Seller \mapsto Buyer: quote[ in ID, in item, out price, out confirmed ]
  Seller \mapsto Buyer: quote[ in ID, in item, in price, out confirmed ]
\}

▶ The \textbf{BUYER} or the \textbf{SELLER} may determine the binding
▶ The \textbf{BUYER} has first dibs in this example
Shopping Cart

Shopping Cart 

def role B, S
parameter out ID key, out lineID key, out item, out qty, out price, out finalize

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

S \rightarrow B: total [in ID, out sum]
B \rightarrow S: settle [in ID, in sum, out finalize]
B \rightarrow S: discard [in ID, out finalize]

}