LoST: Local State Transfer
An Architectural Style for the Distributed Enactment of Business Protocols

Munindar P. Singh
North Carolina State University

October 2011
Interactions and Protocols
Distributed systems of autonomous, heterogeneous agents

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

- $\textit{in}$: 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
- $\textit{out}$: Information that is generated by the protocol instances
  - Bindings can be fed into other protocols through their $\textit{in}$ parameters, thereby accomplishing composition
  - A standalone protocol must adorn all its public parameters $\textit{out}$
- $\textit{nil}$: 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 \{ 
  \textit{role} \ Self , \ Other \\
  \textit{parameter} \ out \ greeting \ key \\
\}

Self \rightarrow\ 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 \( \textit{in} \)
- The key of hi is explicit; often left implicit on messages
The *Pay* Protocol

\[
\text{Pay } \begin{cases}
\text{role } \text{Payer}, \text{ Payee} \\
\text{parameter } \text{in } \text{ID key}, \text{ in } \text{amount}
\end{cases}
\]

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

- At most one payM for each ID
- Not enactable standalone: why?
- The key of payM is implicit; could be made explicit
- Eliding \(\text{\`means\`}\) clauses in this paper
The Offer Protocol

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

Buyer \rightarrow \text{Seller: } \text{rfq} [\text{in ID, out item}] \\
\text{Seller} \rightarrow \text{Buyer: } \text{quote} [\text{in ID, in item, out price}] \\

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

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

\[
B \leftrightarrow S: \ rfq [out \ ID, \ out \ item] \\
S \leftrightarrow B: \ quote [in \ ID, \ in \ item, \ out \ price] \\
B \leftrightarrow S: \ accept [in \ ID, \ in \ item, \ in \ price, \ out \ rID] \\
B \leftrightarrow S: \ reject [in \ ID, \ in \ item, \ in \ price, \ out \ rID] \\
\}
\]

- 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  

  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 Enactment as a Vector of Local Histories

Buyer

rfq

↓

ID, item

quote

↓

ID, price

accept

↓

ID, address

deliver

↓

ID, item, address, outcome

Seller

rfq

↓

quote

↓

ID, price

accept

↓

ID, address

Shipper

ship

↓

ID, item, address

ship

↓

ID, item, address

deliver

↓

ID, item, address, outcome

singh@ncsu.edu (NCSU)
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

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 \( \textit{in} \) parameters
  - No bindings for \( \textit{out} \) and \( \textit{nil} \) parameters exist
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

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

S $\mapsto$ Shipper: ship [in ID, in item, in address]
Shipper $\mapsto$ 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, item, price, outcome

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

  Shipper → B: deliver[in ID, item, address, 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 \( \text{in} \) parameter occurs prior to the message
  - Any \( \text{out} \) parameter occurs simultaneously with the message
- Information reception (receiver’s view)
  - Any \( \text{out} \) or \( \text{in} \) 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
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 \textit{out}
- \textit{Conflict}. At least two competing messages occur
- \textit{Safety} iff the causal structure $\land$ 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
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></th>
<th>ReST</th>
<th>LoST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Modality</strong></td>
<td>Two-party; client-server; synchronous</td>
<td>Multiparty interactions; peer-to-peer; asynchronous</td>
</tr>
<tr>
<td><strong>Computation</strong></td>
<td>Server computes definitive resource state</td>
<td>Each party computes its definitive local state and the parties</td>
</tr>
<tr>
<td></td>
<td></td>
<td>collaboratively and (potentially implicitly) compute the definitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>interaction state</td>
</tr>
<tr>
<td><strong>State</strong></td>
<td>Server maintains no client state</td>
<td>Each party maintains its local state and, implicitly, the relevant</td>
</tr>
<tr>
<td></td>
<td></td>
<td>components of the states of other parties from which there is a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>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>

\[singh@ncsu.edu (NCSU)\]

LoST: Local State Transfer

October 2011
Benefits

▶ Technical
  ▶ Statelessness
  ▶ Consistency
  ▶ Atomicity
  ▶ Natural composition

▶ Conceptual
  ▶ Make protocol designer responsible for specifying causality
  ▶ Avoid contortions of traditional approaches when causality is unclear
Current and Future Directions

▶ Methodology for specifying practical protocols
▶ Expansion of the language to handle role hierarchies
▶ Theoretical results
  ▶ Decision procedures for judging *consistent* enactability
  ▶ Treatment of recursive protocols

http://www.csc.ncsu.edu/faculty/mpsingh/papers/mas/AAMAS-11-IIBIOP.pdf
Thanks!
Properties of Participants

- **Autonomy**
- **Myopia**
  - All choices must be local
  - Correctness should not rely on future interactions
- **Heterogeneity: local ≠ 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
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>Sends in</th>
<th>Sends out</th>
<th>Sends nil</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sends in</td>
<td>Unconstrained</td>
<td>Send out first</td>
</tr>
<tr>
<td>Sends out</td>
<td>Send out first, send at most one</td>
<td>Send nil first</td>
</tr>
<tr>
<td>Sends nil</td>
<td>Receive at least one instance before send</td>
<td>Unconstrained</td>
</tr>
<tr>
<td>Receives in</td>
<td>Receive may occur after send</td>
<td>Send before receive</td>
</tr>
<tr>
<td>Receives out</td>
<td>Receive at least one instance before send</td>
<td>Impossible</td>
</tr>
<tr>
<td>Receives nil</td>
<td>Unconstrained</td>
<td>Unconstrained</td>
</tr>
</tbody>
</table>
Summarizing Approaches for Interaction

<table>
<thead>
<tr>
<th>Declarative (Explicit)</th>
<th>Procedural (Implicit)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Meaning</strong></td>
<td><strong>Operation</strong></td>
</tr>
<tr>
<td>Commitments and other norms</td>
<td>Temporal logic BSPL</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Hard coded within internal reasoning heuristics</td>
</tr>
<tr>
<td></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.
in-out Polymorphism

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

\begin{verbatim}
Flexible-Offer \{ 
  role B, S 
  parameter in ID key, out item, price, out qID

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

  S \mapsto B: quote[ ID, in item, out price, out qID ]
  S \mapsto B: quote[ ID, in item, in price, out qID ]
\}

\end{verbatim}

\begin{itemize}
  \item The price can be adorned \texttt{in} or \texttt{out} in a reference to this protocol
\end{itemize}
The Bilateral Price Discovery protocol

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

  Taker ← Maker: priceRequest[ out reqID, out query ]
  Maker ← Taker: priceResponse[ in reqID, in query, out result ]
}
The Generalized Bilateral Price Discovery protocol

GBPD \{ 
\begin{align*}
  \text{role} & \quad T, \ M \\
  \text{parameter} & \quad \text{reqID}, \ \text{key}, \ \text{query}, \ \text{res} \\

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

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

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 { 
    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 \{ 
\text{role} \text{ Buyer, Seller } \\
\text{parameter} \text{ in ID key, out item, out price, out confirmed} \\
\}

Buyer \mapsto \text{ Seller: rfq [in ID, out item, nil price] } \\
Buyer \mapsto \text{ Seller: rfq [in ID, out item, out price] } \\

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

\begin{itemize}
  \item The \text{ BUYER} or the \text{ SELLER} may determine the binding
  \item The \text{ BUYER} has first dibs in this example
\end{itemize}
Shopping Cart

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

  B \mapsto S: \text{create}[\text{out ID}]
  S \mapsto B: \text{quote}[\text{in ID}, \text{out lineID}, \text{in item}, \text{out price}]
  B \mapsto S: \text{add}[\text{in ID}, \text{in lineID}, \text{in item}, \text{out qty}, \text{in price}]
  B \mapsto S: \text{remove}[\text{in ID}, \text{in lineID}]

  S \mapsto B: \text{total}[\text{in ID}, \text{out sum}]
  B \mapsto S: \text{settle}[\text{in ID}, \text{in sum}, \text{out finalize}]
  B \mapsto S: \text{discard}[\text{in ID}, \text{out finalize}]
}
Macro
- Expanded into the body of a composite protocol: partially enactable
- Maximize concurrency

Procedure
- All or none
- Enable compositionality

BSPL treats references as both
- Enactment is maximally concurrent, at the level of individual messages
- Atomicity avoids undesirable outcomes
ACID Properties
With inspiration from database transactions though with modifications

- Atomicity: if a protocol completes, each reference within it that is initiated also completes
  - Ensured by placing one agent in charge of each conflict
- Consistency: at most one of a set of conflicting references occurs
  - Ensured by placing one agent in charge of each conflict
- Isolation: separate enactments do not interfere
  - Ensured by keys
- Durability: any enactment is permanent
  - Ensured by the immutability of bindings