LoST: Local State Transfer

An Architectural Style for the Distributed Enactment of Business Protocols

Munindar P. Singh
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA
singh@ncsu.edu

***Abstract*—Local State Transfer (LoST) is a simple, declarative approach for enacting communication protocols. LoST is perfectly distributed and relies only upon the local knowledge of each business partner. It involves a novel treatment of the information bases of protocols, especially in terms of how their parameters are specified. As a result, LoST can capture subtle patterns of interaction that more complex approaches cannot handle well. Further, LoST lends itself to implementations that are robust against unordered and lossy message transmission.**

***Index Terms*—Business protocols; messaging; architecture**

I. Introduction

We address the challenge of engineering service-based systems, such as might support cross-organizational business processes [1]. Such systems are characterized by the autonomy and heterogeneity of their participants, which we can naturally model as *agents*. To support loose coupling and arms-length relationships in such settings, it is therefore natural that we would emphasize the *interactions* (specifically, communications in units of messages) among such participants and deemphasize their internal construction [2], [3].

The importance of interaction is increasingly being recognized in industry and academia. Indeed, approaches such as the choreography languages, e.g., WS-CDL [4], attempt to provide a way for specifying the desired interactions. Industry standards in finance (TWIST [5]), health care (HL7 [6]), and manufacturing (RosettaNet [7]) also address interactions as crucial, though often in an over-constrained way—for example, RosettaNet considers only two-party interactions, each of them structured as a request followed by a response.

However, a significant problem with existing approaches to interaction is that they are focused on procedural abstractions, usually based on the same constructs (e.g., sequence, branch, and join) as are used for specifying the internal implementation of a participant. Procedural approaches typically yield specifications that tightly couple the implementations of the participants. Recognition of the above shortcomings has inspired recent research on the notion of business protocols wherein the *business meanings* of the communications among the parties are specified declaratively, usually involving concepts such as commitments, and eschewing operational constraints such as on message ordering and occurrence [8]. The idea is that such declarative specifications would be correct and yet flexible and general—particular enactments would be generated from them, instead of having been (nearly) explicitly enumerated.

However, meaning-based works are usually not well-related to the challenges of real-life distributed computing such as asynchrony, though there is progress [9]. Moreover, stating the meanings relies upon details such as of message ordering and occurrence, especially to establish the interoperability of the participants. Further, the meaning-based approaches usually produce specifications that are confined to the propositional cases because the challenges of dealing with parameterized specifications have not yet been fully understood.

The foregoing situation, with shortcomings of both the traditional and meaning-based approaches, has led us to make a fresh start, in two major parts, on the formulation of the notion of protocol. The first part is a declarative language for specifying the operational aspects of a protocol, dubbed the *Blindingly Simple Protocol Language* or *BSPL*. We previously introduced the main principles motivating BSPL, described the intuitions underlying its design, and provided several important patterns for expressing protocols in BSPL [10].

This paper is about the second part, an architectural framework for distributed applications that we dub *Local State Transfer* or *LoST*. The motivation behind LoST is to provide a way to realize protocol interactions in declarative terms, taking into account the special challenges of distributed computing. Our present contribution is not tied to BSPL, but BSPL provides a convenient vehicle for showcasing it, so we introduce just enough of BSPL to explain our running example.

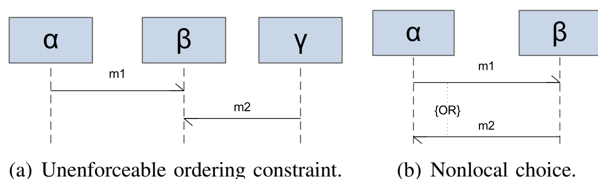
LoST exhibits the philosophy of *interaction-oriented programming* or IOP. IOP involves engineering systems of *agents* or autonomous and heterogeneous participants by describing their interactions. A *protocol* specifies interactions abstractly between two or more *roles*, each to be adopted by an agent for an enactment. The differentiating idea of IOP that LoST adopts is that it treats the interactions themselves as first-class entities. Section IV-B returns to this point.

Our main contribution here is the introduction of the LoST architectural framework and a reference architecture for realizing LoST. LoST maximizes flexibility of the interactions while ensuring correctness constraints, supporting robust enactment, and facilitating local monitoring. This paper is limited to conceptualization and architecture and elides theoretical themes.

II. Motivating LoST

Our interest lies in application-level and especially cross-organizational business communication protocols [2].

As a way to motivate LoST, it is instructive to see how protocols are specified today. A natural way is using message sequence charts (MSCs), similar to UML Sequence Diagrams [11]. Figure 1 shows two MSCs: the protocol roles correspond to the lifelines; each edge connecting two lifelines indicates a message from a sender to a receiver role. Time flows downward by convention and the ordering of the messages is apparent from the chart. MSCs provide primitives to group messages into blocks and to express blocks that are enacted as alternatives of each other, in parallel with each other, or iteratively. MSCs thus provide a rich language in which to specify the control structure of a protocol.



(a) Unenforceable ordering constraint. (b) Nonlocal choice.

Fig. 1. Example challenges for message sequence charts [12].

However, such a control structure might not be possible to enact. For example, Figure 1(a) specifies that message m_1 precede message m_2. But, as far as roles α and γ can discern, m_2 may equally well precede m_1 at role β. Desai and Singh [12] enumerate all the possible situations with precedence constraints between two messages. Figure 1(b) demonstrates the challenge of nonlocal choice, wherein the correctness of a decision by a role depends upon the decisions of other roles. Here, we specify that exactly one of α and β may send a message. But obviously, there is no way for α or β to know if the other role is sending a message. Thus the protocol of Figure 1(b) is not enactable.

Challenges such as the above arise with other notations as well. The usual response is to define criteria for evaluating whether the specification is enactable: a huge body of work addresses such problems, e.g., to determine whether communicating automata can realize a given protocol [13].

Traditional, procedural approaches suffer from two main shortcomings. First, because they lack an information model, they require modelers to state ordering and occurrence constraints in an ad hoc manner. Thus the designs are often fragile: generally over-constrained, but equally as likely lacking important constraints. BSPL avoids this shortcoming by removing the need for such constraints except what are motivated by, and apparent in, the information model. Second, traditional approaches provide no support for enacting protocols: thus, they require procedurally constrained agent adapters that ensure that agents behave correctly, which tightly couples each agent implementation not only to its adapter but also to the implementations of other agents. Thus, the resulting enactments are rigid. LoST avoids this shortcoming by providing constraints that sustain the BSPL information model.

A. Requirements on an Architecture for Distributed Enactment

Our main requirements are that an architectural framework for the distributed enactment of protocols *must* support

autonomy, i.e., systems whose constituent entities are autonomous agents. The framework cannot force an autonomous agent to send any messages. The parties are peers and none of them forces any action by another.

asynchrony, i.e., protocols being enacted over real-life systems that employ asynchronous communications. In particular, it is normal for distributed agents to observe different sets of messages and for two communicating agents to observe the same messages in different orders. The correctness of an outcome should not depend on the relative rates of execution of two agents or the ratios of the various communication channel latencies.

myopia, i.e., support protocols being enacted by *myopic* agents, who work based on what they happen to know at a given moment, do not look ahead, and cannot be counted on to prevent invalid computations. The framework should locally prevent sending messages that could lead to an invalid computation.

consistency, i.e., ensure that the participants view an ongoing interaction consistently. In any snapshot of the system, the local states of the agents can never disagree on any information that they possess, though because of message loss or delay, they may not possess the same information.

B. Principles of LoST

We now formulate the principles that help LoST address the above requirements and which distinguish LoST from other architectural approaches.

No global state. LoST needs no global repository of state. LoST concentrates exclusively on the state of an interaction, which we can think of the *social state* [14] of the parties involved. The social state progresses over the course of the interaction and all information relevant to the social state is explicit within the interaction—in the values of the parameters of the messages exchanged.

Local but not internal. LoST concerns interactions exclusively. Thus, no agent's internal state or business logic are relevant. Further, each agent is limited to knowing what it can observe of the distributed system through the messages it sends and receives. Thus, the correctness or otherwise of an enactment depends solely upon what is locally known and feasible for the agents involved. Each agent maintains its local state, which corresponds to its local view of an ongoing interaction. The purpose of communications is to transfer the relevant components of a local state to another agent, thereby advancing the computation (and justifying the name *LoST*).

Causality through information. The progress of an interaction depends on the causal dependencies among its components. Further, all such dependencies are explicit in the underlying flow of information. The causality applies both (1) between messages and (2) between agents, who enact such messages based on their knowledge.

Robustness. LoST relies directly only of the contents of the messages, not on their ordering. As a result, it is naturally robust against any differences in message order as might be observed by different participants. In particular, LoST can work over non-FIFO channels naturally, including random-access channels where the receiver can read incoming messages in any order (after they have been sent, of course). LoST is also naturally resilient to message loss, and can work over lossy channels provided the repeated retransmission of a message eventually succeeds.

Keys and immutability. BSPL treats each message schema as having a *key*: LoST ensures messages are unique with respect to the stated key. Further, LoST ensures each message is *immutable*: once set, each parameter binding (with respect to a key) never change in any local state.

III. Realizing LoST: Local State Transfer

LoST addresses two crucial aspects of correctness that current frameworks do not, namely, the *causal* structure of a distributed computation and the *knowledge* of the participating agents. The very idea of a traditional control structure (such as sequencing) is based on a centralized way of thinking. In an open setting, the only control that is feasible is one based on the flow of information between the participating agents: all enactable ordering constraints fall out for free from precisely capturing this flow. Also, the challenge of (nonlocal) choice can be stated and addressed precisely in terms of the integrity of the information.

LoST maintains separate local information for each role and ensures that each state locally satisfies constraints on knowledge with respect to parameter adornment.

Listing 1 introduces BSPL [10] via an example. A protocol declaration begins with a name followed by sets of its roles (B, S, SHIPPER) and public parameters (ID, item, price, outcome). The idea is that these roles would be adopted by agents (standing in for business partners) who carry out a purchase interaction. The public parameters describe what information each enactment of *Purchase* involves. BSPL requires that we specify a key, here ID, which determines the unique enactments. Although not explicitly marked in each message, this key applies (and some key must apply) for each message.

Listing 1. 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 response]
 B ↦ S: reject[in ID, in item, in price, out
     outcome, out response]

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

For our present purposes, we do not exercise BSPL's functionality to compose protocols, although that is an important facility for specifications in general. Accordingly, *Purchase* consists of messages *rfq*, *quote*, *accept*, *reject*, *ship*, and *deliver*, each with its sender, receiver, and parameters as specified. During enactment, each parameter is bound to a value. Some parameters that show up on the messages are not public parameters of *Purchase*; these private parameters (address and response) enable intermediate computations.

Listing 1 adorns each parameter with ⌜in⌝ or ⌜out⌝. An ⌜in⌝ parameter on a message is one that is conceptually an input, so its binding must be locally known to the sender prior to sending the message. An ⌜out⌝ parameter is one that is conceptually an output, so its binding must not be locally known to the sender prior to sending the message. In essence, the binding becomes part of the enactment instance during the interaction, i.e., when it becomes public, not when it is internally computed. This is a consequence of our interaction-oriented approach, with benefits in composition in particular [10]. Thus a message instance with a parameter adorned ⌜out⌝ is causally prior to a message instance with the same parameter adorned ⌜in⌝ (provided they use the same key bindings).

The parameter adornments apply to a protocol as well. Each ⌜in⌝ public parameter (*Purchase* has none) must be ⌜in⌝ throughout the body of the protocol. Each ⌜out⌝ parameter must be ⌜out⌝ on an at least one constituent message. During enactment, a protocol can compute only one binding tuple (based on its key); thus at most one constituent message with the same ⌜out⌝ parameter may be sent. Listing 1 satisfies a correctness condition that in case of such out–out conflicts, the same role decides which path is taken by the enactment. Here, B's decision to *accept* or *reject* determines the path.

A. Enactment via History Vectors

To capture the fundamental distribution of LoST, we model its enactments as arising in a fully distributed manner. Each agent participating in a LoST enactment can be thought of as computing a local *history* of observations, which we consider as a sequence of the messages it sends or receives. An entire LoST enactment is nothing more than a *history vector*, whose elements are the histories of the participating agents.

Valid history vectors are subject to two major requirements. First, a message can be observed by its receiver only if it is also (previously) observed by its sender: this is the fundamental causality constraint of distributed computing [16]. Second, although an agent may receive a message at any time, an agent can send a message only if it knows the bindings of all the ⌜in⌝ parameters involved in that message. LoST supports an asymmetry between ⌜in⌝ and ⌜out⌝ parameters in this regard. For an ⌜in⌝ parameter, the sender must already know the binding as part of its local state—because of some prior observation (message emission or reception). For an ⌜out⌝ parameter, the sender must *not* already know the binding as part of its local state. BSPL supports another parameter adornment ⌜nil⌝, which means that the sender neither knows nor adds a binding for the given parameter to its local state.

LoST handles the ⌜nil⌝ adornment, although for simplicity Listing 1 does not use such an adornment.

Receiving a message with an ⌜out⌝ parameter is not essential to receiving the associated parameter binding, which may be conveyed through another message with the same parameter with an ⌜in⌝ adornment. What the ⌜out⌝ adornment signifies is that the message schema (more generally, protocol) generates a binding. Doing so facilitates compositionality [10].

In essence, the agent *creates* the requisite parameter binding as part of sending the given message. The above is the hallmark of interaction orientation. It is true that an agent would exercise some internal reasoning to compute such a binding, but such a binding becomes part of the local state of the sender only when there is a public event involving it, namely, the public event of sending a message. Once an agent sends or receives such a message, its local state changes: it then knows the given binding and it cannot send another message that binds that parameter differently. The immutability of messages helps ensure integrity in that a protocol enactment would compute at most one value for the binding of any parameter (given the applicable key).

Figure 2 shows a history vector, not an MSC-like specification, showing a possible enactment of *Purchase*. This vector consists of three histories, one for each agent. Each history consists of zero or more messages observed (sent or received) in sequence (top to bottom) by that agent. Conceptually, each message consists of the necessary parameter bindings. Figure 2 also shows (via selected important parameters) the causal relationships between the observations in different histories, omitting the parameter bindings to reduce clutter.

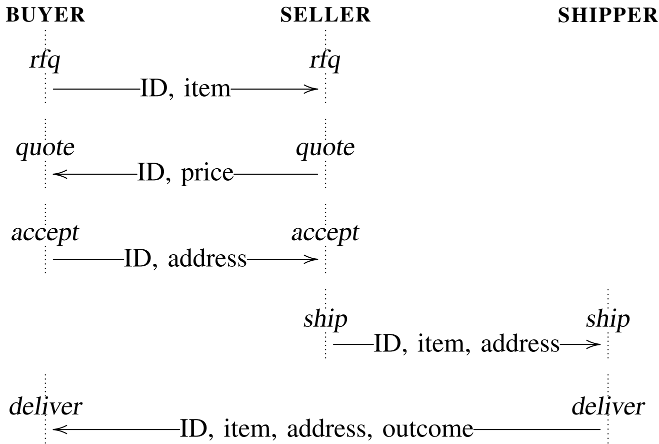


Fig. 2. A possible enactment of the *Purchase* protocol as a history vector.

Two important aspects of distributed enactment that come together in LoST, and which traditional approaches largely disregard, are (1) causality, especially at the level of interactions, and (2) the knowledge of agents. Their interplay is crucial in addressing two important questions that arise in our fully distributed treatment of enactment. First, what history vectors can be generated through LoST? These are all the history vectors that are causally and epistemically sound. Second, how can we ensure that those history vectors correspond to all and only the correct enactments of a protocol? LoST ensures that any (role) history it generates satisfies the information model constraints. For a protocol wherein each nonlocal choice is causally or epistemically preceded by a local choice at some role, all the history vectors produced by LoST are correct.

B. LoST as a Software Architecture

Figure 3 describes LoST schematically as arising between the infrastructure and application layers. LoST maintains and relies upon the public interaction state, which is represented locally at each agent as its knowledge.

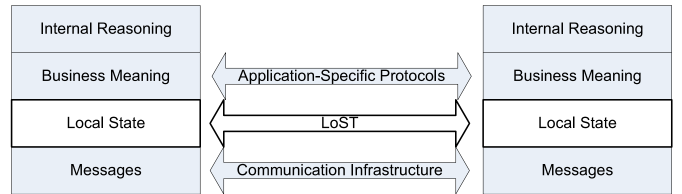


Fig. 3. Understanding LoST schematically as a distinct middleware layer.

LoST is realized as a communication adapter for each agent that presents a simple queue-like interface through which an agent can send and receive messages. LoST naturally works over a communication infrastructure of point-to-point FIFO or non-FIFO queues. LoST relies only upon the local knowledge of an agent to determine whether it can send a message. Additionally, LoST takes advantage of the parameter adornments and their stylized meanings as introduced in Section III-A, to whether a message attempted by an agent is legal.

It is helpful to distinguish structure from meaning in a business protocol. LoST focuses on the structure and distinguishes only between cases where a parameter is bound (if so, consistently across messages) and where it is not bound. For example, in *Purchase*, LoST cares that an *accept* is well-constructed in terms of syntax and local knowledge. The business meaning of *accept*—that it indicates a commitment [1] from the BUYER to pay the SELLER—is important and public, but outside the scope of LoST. LoST captures the data underlying such a commitment as parameter bindings, e.g., the BUYER's commitment to the SELLER will pay $5 or whatever is the binding of price.

In general, an agent's internal reasoning relies upon the specific bindings of the parameters, e.g., to determine what messages with what bindings to send and what internal actions to take. For example, the SELLER would decide what price to *quote* and the BUYER whether to *accept* it. The internal reasoning would generally be supported by the business meaning.

C. Information Model Underlying LoST

We model each protocol as corresponding to exactly the set of interactions it allows, where each interaction corresponds to a possible binding of its parameters. Thus, we imagine a *relation* (i.e., a table) [15] for each protocol (including a message schema) with a tuple for each enactment. Each

protocol defines a *key* consisting of a subset of the parameters declared in it. A set of enactments corresponds to a relation instance (the set of tuples in a relation): thus, a protocol may be instantiated multiple times, limited only by its key.

A key parameter (i.e., part of the key) must not be ⌜nil⌝. As in the relational model, each non-nilable parameter in an enactment (viewed as an instance or tuple of the protocol's relation) must obtain a binding for an enactment to be complete. An important integrity requirement on our information models is that if two constituent message schemas (or protocols) of a protocol share a nonkey parameter, the constituent schemas also share the key of the nonkey parameter, i.e., all the parameters uniquify that parameter. In BSPL, these are the keys of the schemas.

Because all interactions are grounded in messages, we materialize only the relations for the message schemas. Further, only the sender and receiver of a message schema need maintain a relation for it. The relation for a protocol is not materialized. It is simply the universal relation, i.e., the cross-product of the relations of its constituent messages, with exactly one column for each role and parameter (since LoST relies upon parameter names to prevent spurious enactments), and the private roles and parameters projected out. Therefore, the key of the protocol is the union of the keys of its messages. The key restriction is essential for ensuring that the integrity of a protocol enactment is preserved when its constituent messages are properly enacted.

We make the simplifying assumption that the message schemas are in Boyce-Codd Normal Form, which states that there are no functional dependencies in a relation other than from its keys [15]. Thus, the key of a constituent message must be the intersection of the parameters of the message with the parameters in the key for the protocol declaration.

LoST disregards duplication, just as the relational model does. In other words, our information model must define a message schema that allows whatever parts we wish to repeat [10]. LoST simply ensures any enactment respects the stated key. The maximum number of instances of a message schema depend on the domains of its key parameters. Beyond the key constraint, it is up to the sender of a message to decide whether and how to repeat parts of it.

D. Ordering and Occurrence

The effect of parameter bindings is to generate ordering and occurrence conflicts. Receiving a message is always permitted in LoST. As a simple example, suppose some message schemas involve the same key parameter that is ⌜in⌝ along with the same parameter p with different adornments. Table I captures the constraints on sending an instance of any of these schemas relative to sending or receiving another instance of any of these schemas.

Notice that LoST allows a message with p adorned ⌜out⌝ to arrive even when a binding for p (with the same key) is already known to the receiver. This is because LoST does not presume ordered message transmission. Figure 4 shows a simple commerce example where the SELLER receives an

TABLE I
SEND-SEND AND SEND-RECEIVE CONSTRAINTS ON AN AGENT.

	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

order (with ⌜out⌝ price) after the *funds* (with ⌜in⌝ price). Here *order* is not superfluous as it produces the binding of price.

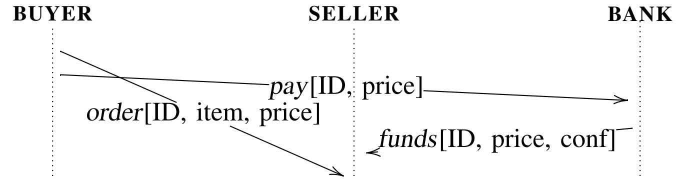


Fig. 4. Example enactment: *funds* arrives before its causally prior *order*.

E. Implementing a LoST Adapter

A LoST adapter for an agent reads, potentially in any order, from the its incoming queue and writes (sends messages) to the queues for any agents that it interacts with. The adapter maintains the relations that capture the local state of its agent. It provides the relevant information to the agent's internal reasoner (possibly via the business meaning component).

Algorithm 1 specifies how a LoST adapter inserts a message into a local store: this algorithm is common to both receiving and sending. An insertion may fail if the message does not match its schema, has an undefined key parameter, is a duplicate, or its contents are inconsistent with a previously stored message of the same or a different name. Discarding duplicates separates the reception of messages from the transfer of their contents, and precludes the possibility of the agent's internal reasoning carrying false dependencies on duplicated messages. Here, [ ] indicate projection.

Algorithm 2 specifies that an agent's LoST adapter attempts to insert an incoming message and, if successful, notifies the agent. Algorithm 3 describes that a sending agent's LoST adapter verifies that the binding specified for each of the ⌜in⌝ parameters is exactly as known to the agent and that no binding is known for each of the ⌜out⌝ or ⌜nil⌝ parameters.

As an example, let us consider the enactment of Figure 2. In *Purchase*, the SELLER sees all messages except *deliver*; the BUYER all except *ship*, and the SHIPPER only *ship* and *deliver*. Their LoST adapters maintain the appropriate relations for each. Initially, all relations are empty and no message with an ⌜in⌝ parameter may be sent. An *rfq* may be sent by the BUYER

Algorithm 1 Inserting a message instance t of schema m.

1: **if** $R(m)$, the relation for schema m, does not exist **then**
2: Throw undefined-message exception
3: **else if** any key or non-nilable parameter of m has nil binding in t **then**
4: Throw schema-violation exception
5: **else if** $t \in R(m)$ **then**
6: **return** false
7: **else**
8: **for all** schemas n **do**
9: $U \leftarrow m \cap n$
10: $K \leftarrow$ subset of U that are key parameters
11: **if** $t[K] \in R(n)[K]$ and $t[U] \notin R(n)[U]$ **then**
12: Throw inconsistent-message exception
13: **end if**
14: **end for**
15: Insert t in $R(m)$
16: **return** true
17: **end if**

Algorithm 2 Receiving a message instance t of schema m.

1: **if** (Insert t into local store) **then**
2: Forward t to the recipient's listener
3: **else** {COMMENT: the insertion failed}
4: Notify sender of t of any exceptions
5: **end if**

if it generates an ID and item. When the message arrives at SELLER, the SELLER may generate a price internally and send a *quote*. Upon its receipt, the BUYER may send *accept* or *reject*: whichever of these messages is first succeeds and inserts a binding for response in the local relation, thereby disabling the other. Such disabling helps LoST ensure the consistency of an enactment. The SELLER needs to know the address to send *ship*: it knows the address only if the BUYER sends it an *accept*. The SHIPPER can send *deliver* only if it receives the bindings for ID, item, and address. Thus, *Purchase* either concludes with the BUYER sending *reject* or the SHIPPER sending *deliver*, each of which produces outcome.

F. Robust Implementation

LoST is naturally insensitive to message transmission order. Further, in general, in normal enactment, a receiver may learn of a parameter binding via multiple paths. There is a crucial distinction between business and technical messages. LoST guarantees that no *business* message is duplicated: each adapter verifies the restrictions on knowledge and conflict, and suppresses duplicate copies of incoming or outgoing messages. Message duplicates have no impact on their local states. The effect of all the parameter adornments is completely captured by the first copy of any message instance.

However, LoST can readily be implemented over *infrastructure* that may arbitrarily retransmit any *technical* message. As long as at least one transmission of each business message succeeds, the overall enactment progresses correctly. For example,

Algorithm 3 Sending a message instance t of schema m.

1: **for all** parameters p in schema m **do**
2: known $\leftarrow$ false
3: **for all** relations $R(n)$ of any schema n where $p \in n$ **do**
4: $U \leftarrow m \cap n$ (COMMENT: thus $p \in U$)
5: **if** $t[U] \in R(n)$ **then**
6: known $\leftarrow$ true
7: **end if**
8: **end for**
9: **if** not known and p is adorned ⌜in⌝ in m **then**
10: Throw in-adornment-violation exception
11: **else if** known and p is adorned ⌜out⌝ in m **then**
12: Throw out-adornment-violation exception
13: **else if** known and p is adorned ⌜nil⌝ in m **then**
14: Throw nil-adornment-violation exception
15: **end if**
16: **end for**
17: **if** (Insert t into local store) **then**
18: Forward t to its recipient
19: **else** {COMMENT: the insertion failed}
20: Notify sender of t of any exceptions
21: **end if**

we may configure the infrastructure to send two copies of each technical message or repeatedly resend technical messages until an acknowledgment to achieve the desired reliability. The immutability of parameter bindings supports a relaxation of message order, and thus facilitates even such naïve approaches being effective. A more sophisticated approach could analyze the protocol to recognize that some messages are implicit acknowledgments of others: for example, once a *quote* arrives, there is no point in repeating *rfq*.

G. Verifying Termination

LoST does not require that any individual agent be aware that a protocol enactment has completed. In Listing 1, the BUYER (B) has visibility into all the parameters declared in *Purchase*. However, in the case of *accept*, *Purchase* terminates when *deliver* occurs, but the SELLER would have no knowledge that *deliver* occurred. If necessary, one can define an alternative protocol in which the BUYER or SHIPPER inform the SELLER of the delivery. Conversely, in the case of *reject*, not informing the SHIPPER of the termination sounds normal.

The above illustrates a strength of LoST: it does not gratuitously couple the agents to each other. However, if you like, you can design a protocol wherein a particular role gathers up all the results. In general, we expect minimal benefit from introducing such a central role into a distributed computation, unless there is an external regulatory reason for doing so.

IV. Major Related Approaches

The immutability of parameter bindings with respect to a key means that parameter bindings can be cached wherever needed and for as long as needed. Further, immutability is

a basis for the robustness of LoST, because there is no harm from multiple emissions and receptions of a message instance. Immutability provides robustness against asynchrony, because it ensures that bindings are never ambiguous or out of date.

A. LoST versus ReST

In broad terms, LoST shares with ReST [17] the philosophical attitude of decoupling components. However, LoST addresses a different problem than traditional web applications. In particular, LoST considers distributed protocols with many loci of enactment as opposed to the client-server, request-response nature of ReST. We can think of LoST as ReST done right for distributed settings. A loose analogy with ReST is that LoST distinguishes adornments of parameters just as ReST distinguishes verbs such as POST, GET, and others.

However, LoST has significant differences that help it avoid some of the main shortcomings of ReST with respect to cross-organizational protocol modeling and enactment. Specifically, ReST's client-server nature maps to orchestration where the client carries out a workflow among the servers. ReST is concerned with side-effects (such as through a POST) and provides no guarantees about the repeated submissions of a POST. LoST deals with immutable bindings and guarantees that repetitions of messages with the same parameters are harmless. Table II summarizes the major differences and similarities between LoST and ReST.

B. LoST versus WS-CDL and BPMN

LoST (and BSPL) take a significantly different philosophical and technical stance from previous languages such as WS-CDL and, especially, BPMN. The most significant differences are the explicit causality and the information centrism of LoST, yielding decentralization and loose coupling.

BPMN is the Business Process Modeling Notation, a standard for business process specifications [18]. BPMN defines *pools* that correspond to BSPL roles. BPMN mainly specifies the operational details of a process using procedural abstractions. BPMN's architecture is of a centralized process engine that has complete knowledge and control over the process. In this way, BPMN violates the autonomy of the participants. BPMN includes the notion of a definitional collaboration whereby one can specify a process in terms of all its collaborations. This idea seems to limit extensibility and preclude compositions, wherein one protocol role is identified with roles in additional protocols [1]. BPMN assumes values can be read from a global context, which does not exist in LoST.

BPMN defines a collaboration as specifying a public process consisting of the interactions or touch-points between participants (p. 24). It defines a choreography as the procedural contract between participants (p. 25) and a conversation as an informal description of message exchanges (p. 26). BPMN uses the traditional procedural constructs (split, join, and so on) for each of the above. It does not specify the messages in a declarative manner as BSPL does, and lacks a semantic characterization of conversations and choreographies.

WS-CDL is smaller than BPMN but quite complex compared to BSPL. WS-CDL's underlying architecture is distributed, like LoST. However, there are significant differences.

First, WS-CDL places into a choreography actions that would be private to an agent, such as what it should do upon receiving a message. LoST, because of its declarative nature, considers only the publicly visible actions and disregards internal reasoning. Doing so maximizes autonomy and heterogeneity. Second, for nested choreographies, WS-CDL relies upon local decision-making by an agent, such as whether to forward a request received in one choreography to another. In LoST, any such composition is accommodated through the common parameters, not through constraints on local decision-making. Third, and importantly, LoST supports a declarative notion of the correctness and completion of a protocol enactment, which WS-CDL lacks. Further, LoST makes the causal dependencies among messages explicit, thereby facilitating more flexible enactments than WS-CDL can. A consequence is that, when two or more messages are performed within a given WS-CDL choreography, they are handled sequentially by default, as in an MSC, whereas LoST omits any arbitrary orders.

Notice that the first two points above indicate that WS-CDL gives first-class status to agents, not to interactions.

C. Additional Relevant Literature

Traditional work on service composition primarily considers orchestrations where a conceptually central party controls two or more services. Semantic web services approaches formalize service behaviors so as to enable planning and constraint reasoning for composition. They give primacy to services, not to their interactions. However, similar techniques, e.g., [19], may be expanded and applied in our setting.

Recent work on *artifacts* treats them as representing a business process that encapsulates all the relevant data of an enactment [20]. In contrast, we separate and focus on the interaction as opposed to the internal reasoning, further separating the interaction into structure (this paper) and meaning (treated as commitments over the parameter bindings [8], [9]). The distributed computing literature has addressed composition [21], [22], but it is focused on capturing internal events as well as inputs and outputs of processes. In our terms, such work treats agents, and not interactions, as first-class entities. Further, we develop a declarative, information-based approach in contrast with the procedural characterization of I/O automata [22].

V. Discussion

The main motivation underlying LoST is to have the agents send and receive messages in a minimally constrained manner. Specifically, LoST enforces the requirements of (1) knowledge, (2) keys, and (3) causality. Each message emission and reception affects the local knowledge of the agent. LoST determines whether each emission is viable based on the local knowledge of its sender at the time of emission and taking into account the parameter adornments. Thus it preserves uniqueness as well as key integrity—as one would expect.

TABLE II
COMPARING LOST AND REST WITH RESPECT TO IMPORTANT FEATURES.

	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
Transfer	State of a resource, suitably represented	Local state of an interaction via parameter bindings, suitably represented
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
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

LoST meets the requirements of Section II-A naturally. It preserves the *autonomy* of the participants constrained only by the protocol specification. LoST never forces an agent to send a message. It allows an agent to send any message that respects its schema and the knowledge requirements of the parameter adornments. LoST operates over infrastructure that is *asynchronous* as well as lossy and able to retransmit messages in an arbitrary manner. LoST works on an agent's current state. For protocols with local treatment of choices, LoST handles *myopia*. LoST guarantees that the knowledge of any two participants is always *consistent*, though not identical because of delay or loss.

We should emphasize that LoST does not impose any additional overhead in terms of storage. In today's common practice, all businesses already log all their business messages. They do so for managing their business operations, for regulatory reasons, and to support analytics and optimization—usually all three. LoST verifies that the messages are consistent and unique, so there is an overhead for such tests but it yields benefits in improved flexibility and robustness.

The idea of correlating transactions or activities within a workflow is well known. Keys as used in LoST are a natural enhancement of that idea: we apply keys on interactions and use them as a basis for supporting immutability and preventing spurious duplication.

LoST provides the architectural underpinnings of BSPL, which is a simple declarative approach for expressing communication protocols based on two main constructs: a way to specify a message and a way to compose protocols [10]. BSPL protocols can be naturally enacted via LoST. In particular, BSPL lifts the guarantees of LoST, such as their uniqueness and consistent views, to protocols at large.

A useful future direction is enhancing the treatment of the LoST information model. For instance, it would be appropriate to entertain multiple keys for a protocol. Further, it would be useful to understand the impact of asserting explicit functional dependencies [15] on the potential enactability of protocols, and so on. Some natural extensions that we will be considering include a principled treatment of multicast (where multiple agents playing the same role receive a message), accommodating discovery protocols (where the roles are bound late during enactment), and recursive protocols.

ACKNOWLEDGMENTS

This work was partially supported by the OOI Cyberinfrastructure program, which is funded by NSF contract OCE-0418967 with the Consortium for Ocean Leadership via the Joint Oceanographic Institutions. Thanks to Matthew Arrott, Amit Chopra, and the reviewers for helpful comments.

REFERENCES

[1] N. Desai, A. K. Chopra, and M. P. Singh, "Amoeba: A methodology for modeling and evolution of cross-organizational business processes," *ACM TOSEM*, vol. 19, no. 2, pp. 6:1–6:45, Oct. 2009.
[2] C. Bussler, "The role of B2B protocols in inter-enterprise process execution," in *Proc. Technologies for E-Services*, ser. LNCS, vol. 2193. Berlin: Springer, 2001, pp. 16–29.
[3] B. Medjahed, B. Benatallah, A. Bouguettaya, A. H. H. Ngu, and A. K. Elmagarmid, "Business-to-business interactions: issues and enabling technologies," *VLDB Journal*, vol. 12, no. 1, pp. 59–85, May 2003.
[4] WS-CDL, "Web services choreography description language version 1.0," Nov. 2005, www.w3.org/TR/ws-cdl-10/.
[5] TWIST, "Twist wholesale financial markets standard," Nov. 2008, http://www.twiststandards.org/ index.php? option=com_content & view=article & id=75 & Itemid=112.
[6] HL7, "Health Level Seven," 2007, http://www.hl7.org.
[7] RosettaNet, "Home page," 1998, www.rosettanet.org.
[8] M. P. Singh, A. K. Chopra, and N. Desai, "Commitment-based service-oriented architecture," *IEEE Computer*, vol. 42, no. 11, pp. 72–79, 2009.
[9] A. K. Chopra and M. P. Singh, "Multiagent commitment alignment," in *Proc. AAMAS*. 2009, pp. 937–944.
[10] M. P. Singh, "Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language," in *AAMAS*. 2011.
[11] M. Fowler, *UML Distilled*, 3rd ed. Addison-Wesley, 2003.
[12] N. Desai and M. P. Singh, "On the enactability of business protocols," in *Proc. AAAI*. 2008, pp. 1126–1131.
[13] X. Fu, T. Bultan, and J. Su, "Conversation protocols," *Theoretical Computer Science*, vol. 328, no. 1–2, pp. 19–37, 2004.
[14] M. Baldoni, C. Baroglio, and E. Marengo, "Behavior-oriented commitment-based protocols," in *Proc. ECAI*, 2010, pp. 137–142.
[15] R. Elmasri and S. Navathe, *Fundamentals of Database Systems*, 2nd ed. Redwood City, CA: Benjamin Cummings, 1994.
[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *CACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
[17] S. Vinoski, "RESTful web services development checklist," *IEEE Internet Computing*, vol. 12, no. 6, pp. 95–96, 2008.
[18] OMG, "Business process model and notation (BPMN), version 2.0 beta," Jun. 2010, Object Management Group. http://bpmn.org/.
[19] M. Burstein, C. Bussler, T. Finin, M. N. Huhns, M. Paolucci, A. P. Sheth, S. Williams, and M. Zaremba, "A semantic Web services architecture," *IEEE Internet Computing*, vol. 9, no. 5, pp. 72–81, Sep. 2005.
[20] D. Cohn and R. Hull, "Business artifacts," *IEEE Data Engineering Bulletin*, vol. 32, no. 3, pp. 3–9, 2009.
[21] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Reading, MA: Addison-Wesley, 1988.
[22] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *CWI Quarterly*, vol. 2, no. 3, pp. 219–246, 1989.