Interaction-Oriented Programming for Decentralized Service Engagements

Munindar P. Singh
(with Amit K. Chopra and Samuel H. Christie V)

North Carolina State University

International Conference on Web Information Systems Engineering (WISE)
October 2021
Agents Helping Principals Exercise Autonomy

Inherently decentralized

- Each agent reflects the autonomy of its principal
- How can we realize a multiagent system based that accommodates the autonomy of its principals?
  - Does not unduly constrain an agent’s decision making
  - Supports flexible interactions
  - Enables loose coupling between agents
Healthcare Application

Patient sends a Complaint to Physician, who sends a Prescription to Pharmacy, who sends Fulfill to Patient

- Autonomy means no one need send any message!
- Three parties, not client server
- Healthcare standards: Health Level 7 (HL7), Integrate the Healthcare Enterprise (IHE)
  - Informally described interactions difficult to implement correctly
Purchase Order (PO) Fulfillment Application

Several items in a PO that may be wrapped and packed independently to create a shipment.
Asynchronous Communication
Without message ordering guarantees from the communication infrastructure!

▶ Today: Commonplace to rely on at least FIFO delivery
▶ Challenge: Coordinate decentralized computation without assuming ordered delivery
Interaction-Oriented Methodology

Lucid hi-fi computational abstractions for engineering sociotechnical systems

- Engineer a system by composing declarative specifications of interactions
  - Strictly without considering agents (endpoint implementations)
- Engineer an agent on the basis of those specifications
  - Strictly without considering other agents
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 strings)

- ▶ ⌜ 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
Protocol in BSPL: Main Ideas

- **Declarative**
  - No control flow, no control state

- **Information-based**
  - Specifies the computation of distributed information object
    - Message specification is atomic protocol
  - Specified via parameters

- **Explicit causality**
  - The messages an agent can send depends upon what it knows
  - Via parameter adornments `out`, `in`, `nil`

- **Integrity**
  - Agent only sends messages that preserve consistency of objects
  - Via key constraints

- **Asynchronous messaging**

- **Requires no ordering from infrastructure**

- **Composition and verification**
The *Initiate* protocol

```java
Initiate {
  role B, S
  parameter out ID key, out item
  B ⇝ S: rfq [out ID key, out item]
}
```
The *Initiate* protocol

**Initiate**

```
Initiate {
    role B, S
    parameter out ID key, out item

    B → S: rfq [out ID key, out item]
}
```

<table>
<thead>
<tr>
<th>B:rfq</th>
<th>S:rfq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID item</td>
<td>ID item</td>
</tr>
</tbody>
</table>
The *Initiate* protocol

\[
\text{Initiate} \{ \\
\text{role B, S} \\
\text{parameter out ID key, out item} \\
\text{B } \leftrightarrow \text{ S: rfq [out ID key, out item]} \\
\}
\]

<table>
<thead>
<tr>
<th>ID</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fig</td>
</tr>
</tbody>
</table>

**Initiate (virtual)**

<table>
<thead>
<tr>
<th>B:rfq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S:rfq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
</tbody>
</table>

The *Initiate* protocol

\[
\text{Initiate} \ \{ \\
\text{role B, S} \\
\text{parameter out ID key, out item} \\
\text{B }\leftrightarrow \text{ S: rfq [out ID key, out item]}
\}
\]
The *Initiate* protocol

\[
\text{Initiate} \left\{ \\
\text{role B, S} \\
\text{parameter out ID key, out item} \\
\text{B} \leftrightarrow \text{S: rfq [out ID key, out item]} \\
\right\}
\]

<table>
<thead>
<tr>
<th></th>
<th>ID</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>fig</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>jam</td>
</tr>
</tbody>
</table>

\[B:rfq\]

<table>
<thead>
<tr>
<th>ID</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fig</td>
</tr>
<tr>
<td>5</td>
<td>jam</td>
</tr>
</tbody>
</table>

\[S:rfq\]

<table>
<thead>
<tr>
<th>ID</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>jam</td>
</tr>
</tbody>
</table>
The *Initiate* protocol

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

B \rightarrow S: rfq [out ID key, out item]
The *Initiate* protocol

```
Initiate {
    role B, S
    parameter out ID key, out item

    B → S: rfq [out ID key, out item]
}
```

**Initiate (virtual)**

<table>
<thead>
<tr>
<th>ID</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fig</td>
</tr>
<tr>
<td>5</td>
<td>jam</td>
</tr>
<tr>
<td>8</td>
<td>fig</td>
</tr>
</tbody>
</table>

B:rfq

<table>
<thead>
<tr>
<th>ID</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fig</td>
</tr>
<tr>
<td>5</td>
<td>jam</td>
</tr>
<tr>
<td>8</td>
<td>fig</td>
</tr>
</tbody>
</table>

S:rfq

<table>
<thead>
<tr>
<th>ID</th>
<th>item</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>jam</td>
</tr>
</tbody>
</table>
The *Offer* Protocol

```plaintext
Offer {
    role B, S
    parameter out ID key, out item, out price
    B → S: rfq[ out ID, out item ]
    S → B: quote[ in ID, in item, out price ]
}
```
The Offer Protocol

Offer {
  role B, S
  parameter out ID key, out item, out price

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

<table>
<thead>
<tr>
<th>Offer (virtual)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B:rfq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B:quote</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S:rfq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S:quote</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
</tbody>
</table>

Singh Interaction-Oriented Programming for Services WISE 2021
The Offer Protocol

Offer {
role B, S
parameter out ID key, out item, out price

B → S: rfq [out ID, out item]
S → B: quote [in ID, in item, out price]
}

<table>
<thead>
<tr>
<th>Offer (virtual)</th>
<th>ID</th>
<th>item</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>B:rfq</td>
<td>1</td>
<td>fig</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B:quote</th>
<th>ID</th>
<th>item</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>fig</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S:rfq</th>
<th>ID</th>
<th>item</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>fig</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S:quote</th>
<th>ID</th>
<th>item</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>fig</td>
<td>10</td>
</tr>
</tbody>
</table>
The Offer Protocol

Offer {
role B, S
parameter out ID key, out item, out price

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

<table>
<thead>
<tr>
<th>Offer (virtual)</th>
<th>ID</th>
<th>item</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>B:rfq</td>
<td>1</td>
<td>fig</td>
<td>10</td>
</tr>
<tr>
<td>B:quote</td>
<td>1</td>
<td>fig</td>
<td>10</td>
</tr>
<tr>
<td>S:rfq</td>
<td>1</td>
<td>fig</td>
<td>10</td>
</tr>
<tr>
<td>S:quote</td>
<td>1</td>
<td>fig</td>
<td>10</td>
</tr>
</tbody>
</table>

Singh Interaction-Oriented Programming for Services WISE 2021 14
The Offer Protocol

Offer {
  role B, S
  parameter out ID key, out item, out price

  B → S: rfq [out ID, out item]
  S → B: quote [in ID, in item, out price]
}

<table>
<thead>
<tr>
<th>Offer (virtual)</th>
<th>B:rfq</th>
<th>B:quote</th>
<th>S:rfq</th>
<th>S:quote</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>item</td>
<td>price</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>fig</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>fig</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>fig</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Singh Interaction-Oriented Programming for Services WISE 2021 14
The *Decide Offer* Protocol
Choice: *accept* and a *reject* with the same ID *cannot* both occur

```plaintext
Decide Offer {
    role B, S
    parameter out ID key, out item, out price, out decision

    B 7→ S: rfq [out ID, out item]
    S 7→ B: quote [in ID, in item, out price]

    B 7→ S: accept [in ID, in item, in price, out decision]
    B 7→ S: reject [in ID, in item, in price, out decision]
}
```
The *Decide Offer* Protocol

Choice: *accept* and a *reject* with the same ID *cannot* both occur

\[
\text{Decide Offer} \{ \\
\text{role } B, \ S \\
\text{parameter out ID key, out item, out price, out decision} \\
B \rightarrow S: \ rfq\ [\text{out ID, out item}] \\
S \rightarrow B: \ quote\ [\text{in ID, in item, out price}] \\
B \rightarrow S: \ accept\ [\text{in ID, in item, in price, out decision}] \\
B \rightarrow S: \ reject\ [\text{in ID, in item, in price, out decision}] \\
\}
\]

<table>
<thead>
<tr>
<th>Decide Offer (virtual)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B:rfq</th>
<th>B:quote</th>
<th>B:accept</th>
<th>B:reject</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>item</td>
<td>ID</td>
<td>item</td>
</tr>
<tr>
<td>1</td>
<td>fig</td>
<td>1</td>
<td>fig</td>
</tr>
</tbody>
</table>
The *Decide Offer* Protocol

Choice: *accept* and a *reject* with the same ID *cannot* both occur

```
Decide Offer { 
  role B, S 
  parameter out ID key, out item, out price, out decision 

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

<table>
<thead>
<tr>
<th>ID</th>
<th>item</th>
<th>price</th>
<th>decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fig</td>
<td>10</td>
<td>nice</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B:rfq</th>
<th>B:quote</th>
<th>B:accept</th>
<th>B:reject</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>item</td>
<td>ID</td>
<td>item</td>
</tr>
<tr>
<td>1 fig</td>
<td></td>
<td>1 fig</td>
<td>10</td>
</tr>
</tbody>
</table>

Singh

Interaction-Oriented Programming for Services

WISE 2021
The \textit{Decide Offer} Protocol

Choice: \textit{accept} and a \textit{reject} with the same ID cannot both occur

\textbf{Decide Offer} \{
\text{role B, S}
\text{parameter out ID key, out item, out price, out decision}

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

\text{B} \leftrightarrow \text{S: accept [in ID, in item, in price, out decision]}
\text{B} \leftrightarrow \text{S: reject [in ID, in item, in price, out decision]}
\}\n
\begin{tabular}{cccc}
\text{ID} & \text{item} & \text{price} & \text{decision} \\
1 & \text{fig} & 10 & \text{nice} \\
\end{tabular}
The *Purchase Protocol*

```
Purchase {
  role B, S, Shipper
  parameter out ID key, out item, out price, out outcome
  private address, resp

  B 7→ S: rfq [out ID, out item]
  S 7→ B: quote [in ID, in item, out price]
  B 7→ S: accept [in ID, in item, in price, out address, out resp]
  B 7→ S: reject [in ID, in item, in price, out outcome, out resp]

  S 7→ Shipper: ship [in ID, in item, in address]
  Shipper 7→ B: deliver [in ID, in item, in address, out outcome]
}
```

- *reject* conflicts with *accept* on resp (a *private* parameter)
- *reject* or *deliver* must occur for completion (to bind outcome)
Standing Order
As in insurance claims processing

Insurance—Claims
{ role Vendor (V), Subscriber (S) parameter out pID key, out cID key, out claim, out response

Create—Policy (V, S, out pID, out details)
S \rightarrow V: claimRequest [in pID, out cID, out claim]
V \rightarrow S: claimResponse [in pID, in cID, out response]
}

- Illustrates composite keys
  - A policy (identified by a binding for pID) may be associated with multiple claims (each identified by a binding for cID)
- Composes protocol Create-Policy, which produces bindings for pID
Realizing BSPL via Local State Transfer (LoST)

Think of the message logs you want

- For each role
  - For each message that it sends or receives
    - Maintain a local relation of the same schema as the message
- Receive and store any message provided
  - It is not a duplicate
  - Its integrity checks with respect to parameter bindings
  - Garbage collect expired sessions: requires additional annotations
- Send any unique message provided
  - Parameter bindings agree with previous bindings for the same keys for \texttt{in} parameters
  - No bindings for \texttt{out} and \texttt{nil} parameters exist
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
  ▶ Parameter has only one value relative to the key

▶ Immutability
  ▶ Durability
  ▶ Robustness: insensitivity to
    ▶ Reordering by infrastructure
    ▶ Retransmission: one delivery is all it needs
Ideal Protocol-Based System Architecture

Principal

Agent

Protocol

Agent

Principal

Asynchronous Communication Infrastructure

- **Constraints**
  1. Agent ensures the correctness of its emissions. To do so, it needs nothing but its local state (history of prior emissions and receptions)
  2. The reception of any message is correct, if it was emitted correctly
  3. Asynchrony: Emissions nonblocking; receptions nondeterministic
     - No ordered delivery guarantee needed from infrastructure.
  4. The protocol is the complete operational specification of the system

- **Assumption**: Infrastructure delivers only sent messages
  - No guaranteed delivery assumed
## Comparing LoST and ReST

<table>
<thead>
<tr>
<th>Modality</th>
<th>ReST</th>
<th>LoST</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computation</strong></td>
<td>Two-party; client-server; synchronous</td>
<td>Multiparty interactions; peer-to-peer; asynchronous</td>
</tr>
<tr>
<td></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><strong>State</strong></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 <code>GET</code></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><code>GET, POST, ...</code></td>
<td><code>⌜in⌟, ⌜out⌟, ⌜nil⌟</code></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>
Decentralized Applications on FaaS Platforms

Protocol + FaaS = highly modular and concurrent agent out of the box
Developer focuses on business logic
Implemented on AWS Lambda

- Decision-making components, Proactor and Reactor
- Many instances of each as FaaS functions
- Each Reactor handles one message schema (encoded in JSON)
- Unique Checker for each agent
- Deployment: configuration file per agent referring to layers for components and resources
Implementation on Amazon Web Services

Using the term *stack* instead of *service* to avoid confusion

- Serverless Framework manages the configuration of multiple resources
- Stack: a declarative specification of resources to be interpreted
  - Locally using Serverless Framework tools
  - Via a cloud deployment system, e.g., CloudFormation
- Separate stack for each agent to avoid coupling them unnecessarily: generally, agents are provided by different organizations
- Map roles to endpoints, as below

```
{
  "Merchant": "https://5yo8ouXXXX.execute-api.us-east-1.amazonaws.com/merchant/messages",
  "Labeler": "https://awj8rrXXXX.execute-api.us-east-1.amazonaws.com/labeler/messages",
  "Wrapper": "https://23y4xcXXXX.execute-api.us-east-1.amazonaws.com/wrapper/messages",
  "Packer": "https://akuf0nXXXX.execute-api.us-east-1.amazonaws.com/packer/messages"
}
```
Layer for the Checker Component

checker:
  path: layer # required, path to layer contents on disk
  name: PoSCheckerLayer # optional, Deployed Lambda layer name
  description: Layer for sharing the PoS checker module # optional, Description to publish to AWS
  licenseInfo: GPLv3 # optional
  package:
    include:
      - '!./**'
      - checker.py

▶ Common component in our architecture
JSON Specification of the *Logistics* Protocol

```
{
  "name": "Logistics",
  "type": "protocol",
  "parameters": ["orderID", "itemID", "item", "status"],
  "keys": ["orderID", "itemID"],
  "ins": [],
  "outs": ["orderID", "itemID", "item", "status"],
  "nils": [],
  "roles": ["Merchant", "Wrapper", "Labeler", "Packer"],
  "messages": {
    "RequestLabel": {
      "name": "Logistics/RequestLabel",
      "type": "message",
      "parameters": ["orderID", "address"],
      "keys": ["orderID"],
      "ins": [],
      "outs": ["orderID", "address"],
      "nils": [],
      "to": "Labeler",
      "from": "Merchant"
    }
  }
}
```
Merchant’s Checker Specification

MerchantChecker:

- **name**: MerchantChecker
- **handler**: /opt/checker.lambda_handler
- **layers**:
  - `{cf:pos-components-dev.CheckerLayerExport}`
  - `{cf:pos-components-dev.DepsLayerExport}`
  - `{cf:logistics-dev.ConfigurationLayerExport}`

reservedConcurrency: 1

Declares a function MerchantChecker, which uses CheckerLayerExport to load the checker code, which includes the Checker, and loads the configuration layer.
Merchant’s Reactor Specification for the *Packed* Message

```python
PackedReactor:
    name: Merchant_Packed_Reactor
    handler: packed_reactor.lambda_handler
    package:
        include:
            - packed_reactor.py
```

- Identifies its Lambda handler
- Identifies the implementation code to load
Merchant Package Specification

package:
  individually: true
  include:
    — '!**'
    — reactors.json

- Registers the Reactor by adding reactors.json for Merchant
- Specifies what to include and exclude
- Merchant’s Reactor mapping is below:

```json
{
  "Logistics/Packed":
    "arn:aws:lambda:us-east-1:834106683512:function:Merchant_Packed_Reactor"
}
```
def lambda_handler(event, context):  # wrapped reactor
    message = event["message"]
    enactment = event["enactment"]
    labeled_msg = next((m for m in enactment if m.get("label")), None)
    if labeled_msg:
        # send packed notification for item
        payload = {
            "type": "send",
            "to": "Merchant",
            "message": {
                "orderID": message["orderID"],
                "itemID": message["itemID"],
                "wrapping": message["wrapping"],
                "label": labeled_msg["label"],
                "status": "packed",
            },
        },
        payload = json.dumps(payload).encode("utf-8")
        print("Sending Packed: {}".format(payload))
        response = client.invoke(
            FunctionName="PackerChecker",
            ...
# Merchant’s Proactor Specification

## Functions

**order:**
- **handler:** `order.writeToDynamo`
- **events:**
  - `httpApi`: POST /orders # to receive customer orders

**package:**
- **include:**
  - `order.py`

**PO_proactor:**
- **handler:** `PO_proactor.get_order_proactor`
- **events:**
  - `stream`:
    - **type:** DynamoDB
    - **arn:**
      - `Fn::GetAtt`: [ordersTable, StreamArn]

**package:**
- **include:**
  - `PO_proactor.py`

► Proactor produces events or respond to outside events
Evaluation on AWS Lambda: Transaction Time and Throughput

- Asynchronously submit 1,000 POs (1–4 items) for *Fulfillment*
- Set all DynamoDB tables to autoscale, with no throttling of requests
- Normal setting: no delay
- Delayed setting: 1s delay to simulate heavier processing
- Delay: analyze the merchant agent’s message timestamps

<table>
<thead>
<tr>
<th>Experiment</th>
<th>PO Duration</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean (s)</td>
<td>St. Dev. (s)</td>
</tr>
<tr>
<td>Normal</td>
<td>266.51</td>
<td>51.45</td>
</tr>
<tr>
<td>Delayed</td>
<td>267.27</td>
<td>46.45</td>
</tr>
</tbody>
</table>

- Reactor without delay takes 1ms to 380ms but delay of 1,000ms has little effect—low effect size of difference (Cohen’s $d = 0.015$)
Evaluation on AWS Lambda: Concurrency
Concurrent instances of Lambda functions via AWS CloudWatch monitoring console

▶ Deserv takes advantage of automatic scaling in FaaS
▶ The Checker and database are potential bottlenecks
▶ The business computation takes place in the Reactor, however

<table>
<thead>
<tr>
<th>Component</th>
<th>Number of Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Normal</td>
</tr>
<tr>
<td>Receiver</td>
<td>2</td>
</tr>
<tr>
<td>Reactor</td>
<td>2</td>
</tr>
<tr>
<td>Emitter</td>
<td>3</td>
</tr>
</tbody>
</table>
Multiple Enactments Are Possible and Desirable

(a) Shipment first
(b) Payment first
(c) Concurrent

Information-based protocol languages
- Declarative: based on causality and integrity constraints
- Produce maximal flexibility compatible with application meaning

Main finding: Information protocols formalized declaratively make possible optimizations that are unavailable for traditional protocol languages
Protocol: Purchase with Cancellation
Causality and integrity are captured via in, out, and nil

```plaintext
protocol PO Pay Cancel Ship {
    roles B, S
    parameters out ID key, out item, out price, out outcome
    private pDone, gDone, rescind

    B \mapsto S: PO [out ID key, out item, out price]
    B \mapsto S: cancel [in ID key, nil pDone, nil gDone, out rescind]
    B \mapsto S: pay [in ID key, in price, in item, out pDone]
    S \mapsto B: ship [in ID key, in item, nil rescind, out gDone]
    S \mapsto B: cancelAck [in ID key, in rescind, nil gDone, out outcome]
    S \mapsto B: payAck [in ID key, in pDone, out outcome]
}
```
Generating a Tableau

- A tableau node captures the enactment so far
- Any enabled emission or reception can take place next
  - Explosion of possibilities; most are irrelevant variants
- Some are relevant alternatives: enabled emissions may become disabled
- All enabled receptions remain enabled (until they occur)

Generate a reduced tableau ⇒
All logically distinct possibilities are retained
Superfluous variants are reduced
Causal Relations Between Messages

Derived entirely from their parameters and how they are adorned

Two messages with a common parameter:

▶ Some message with \( `\text{out} `\) must precede a message with \( `\text{in} `\)
  ▶ Directly endows (necessary precursor): when only one message has \( `\text{out} `\)

▶ Receiving message with \( `\text{in} `\) enables another with \( `\text{in} `\), disables \( `\text{out} `\) and \( `\text{nil} `\)

▶ Sending or receiving message with \( `\text{out} `\) enables another with \( `\text{in} `\), disables \( `\text{out} `\) and \( `\text{nil} `\)

▶ Message with \( `\text{nil} `\) has no effect

Chaining the above:

▶ Enablement: chain of one or more direct enablements

▶ Tangles with: Doesn’t endow (make possible) a message, and disables a message or disables a causal precursor of the message

▶ Incompatible: tangles with and at least one of them an emission
Reduction Method

Sensitive observations (emissions or receptions): if they may disable others or be disabled by others

- Expand tableau using nonsensitive observations in arbitrary order
- If only sensitive observations
  - Produce partitions heuristically (approximate vertex cover)
  - Create one branch for each compatible set in the partition
  - Develop each branch with an arbitrary member of the set
- Iterate until all branches end or hit an inconsistency
To verify a property: assert suitable proposition at root

Stop tableau expansion upon contradiction

Here: complete tableau for illustration

22 nodes instead of 1,495
## Performance Comparison

Major improvements on practical protocols
Protocol listings are online or in works cited from the Tango paper

<table>
<thead>
<tr>
<th>Protocol</th>
<th>No Branch Reductions</th>
<th>Tango</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes</td>
<td>Branches</td>
</tr>
<tr>
<td>PO Pay Cancel Ship</td>
<td>1,495</td>
<td>490</td>
</tr>
<tr>
<td>Block Contra</td>
<td>1,802</td>
<td>612</td>
</tr>
<tr>
<td>Independent</td>
<td>453</td>
<td>90</td>
</tr>
<tr>
<td>NetBill (e-commerce)</td>
<td>4,097</td>
<td>1,246</td>
</tr>
<tr>
<td>HL7 Create Lab Order</td>
<td>59,259</td>
<td>17,814</td>
</tr>
</tbody>
</table>

Tango: Precisely captures tanglements based on information protocols to justify strong optimizations
Discussion

Demonstrate how to achieve decentralization, avoid client-server programming, while taking advantage of serverless computing

- IOP modularizes service systems
  - Separates agents from one another
  - Coupled only to the extent as specified in a protocol
  - Information model supports asynchrony

- Deserv unifies protocol-based programming of service engagements with serverless computing
  - Each agent is a composition of microservices
  - Immutability of information is naturally compatible with functional programming

- Tango exploits causality to reduce complexity of formal verification
Directions and Thanks

- Implementation: https://gitlab.com/masr
- Tools address
  - Formal verification of protocols
  - Fault tolerant agents as microservices
  - Serverless computing
  - IoT applications
  - Blockchain implementation of contracts
- Directions: Enhanced tooling and evaluation
- Collaborators welcome!
- Thanks!
  - US National Science Foundation grant IIS-1908374
  - UK EPSRC grant EP/N027965/1