Deserv: Decentralized Serverless Computing

Samuel H. Christie V
School of Computing and Communications
Lancaster University
Lancaster, UK
ORCID: 0000-0003-1341-0087

Amit K. Chopra
School of Computing and Communications
Lancaster University
Lancaster, UK
ORCID: 0000-0003-4629-7594

Munindar P. Singh
Department of Computer Science
North Carolina State University
Raleigh, USA
ORCID: 0000-0003-3599-3893

***Abstract*—A *decentralized* application involves multiple autonomous principals, e.g., humans and organizations. Autonomy motivates (i) specifying a decentralized application via a *protocol* that captures the interactions between the principals, and (ii) a programming model that enables each principal to independently (from other principals) construct its own protocol-compliant *agent*. An agent encodes its principal's decision making and represents it in the application. We contribute *Deserv*, the first protocol-based programming model for decentralized applications that is suited to the cloud. Specifically, Deserv demonstrates how to leverage *function-as-a-service* (FaaS), a popular serverless programming model, to implement agents. A notable feature of Deserv is the use declarative *protocols* to specify interactions. Declarative protocols support implementing stateful agents in a manner that naturally exploits the concurrency and autoscaling benefits offered by serverless computing.**

***Index Terms*—multiagent systems, protocols, programming model**

I. Introduction

Many applications in important domains such as e-commerce, health, and finance are conceptually *decentralized* because they involve autonomous *principals*, such as humans and organizations. Autonomy means that each principal exercises independent decision making and engages in arms-length interactions with others.

Autonomy motivates a software architecture wherein each principal is represented by a software *agent* that (i) encodes the principal's decision making, and (ii) interacts with other agents via asynchronous messaging. Moreover, this architecture is loosely coupled in that each agent is constructed independently of other agents. Decentralization motivates an application representation that reflects such an architecture. Loose coupling means decentralization remains valuable even when the same principal controls all agents—by enabling incremental maintenance.

Notably, traditional software approaches realize an application as a *unitary* machine (e.g., a Web service) and are therefore inadequate for modeling and implementing decentralized applications [1].

Protocols are crucial to realizing such a decentralized architecture [2]. A *protocol* specifies a decentralized application by specifying the constraints on messaging between agents, but otherwise leaves the principals free to implement their own agents as they desire. A protocol-based *programming model* facilitates the implementation of protocol-compliant agents via concrete programming abstractions that constitute an API [3], [4]. The idea is that if an agent developer follows the model, then the agent is guaranteed to be compliant with the protocol. Of course, the developer is responsible for ensuring the agent does what its principal desires.

With the arrival of serverless computing, cloud computing is moving into a new era. Whereas earlier, cloud computing supported the deployment of containerized applications, with serverless computing, developers use cloud-native programming abstractions to develop applications [5]. The *Function-as-a-Service* (FaaS) programming model is the basis for today's serverless computing. Although still maturing, FaaS is representative of the ambition of the serverless paradigm—abstract away the cloud from application developers as much as possible. Specifically, the ambition is that developers need only focus on writing the application logic, leaving scaling, fault tolerance, and resource management in general to the cloud. FaaS is offered by Amazon AWS Lambda, Microsoft Azure Functions, IBM Cloud Functions, and Google Cloud Functions.

FaaS, however, is a programming model strictly in the traditional mold: It is a programming model for implementing an application as a Web service [6]. The service orchestrates [7] other services and remains a unitary locus of control. Therefore, the abstractions FaaS offers are inadequate for modeling and implementing decentralized applications.

The problem we address is how to build decentralized applications using FaaS in which each constituent *agent* maps to a separate computation; the agents communicate asynchronously and share no storage, reflecting modern thinking on building loosely-coupled systems [8].

We contribute *Deserv*, a programming model for serverless decentralized applications. Deserv enables realizing a decentralized application as a protocol-based multiagent system. A declarative *information protocol* specifies the roles that agents would adopt and the messages between roles [9]. Whether it is correct for an agent to send a message at some point depends solely upon what information the agent has observed in prior messages, whether sent or received. Receptions are unconstrained: an agent can receive any message that has arrived at any point. This means that a protocol can be enacted over infrastructure without requiring ordering guarantees. Further, protocols support maximal concurrency, since each task to be performed or message to be sent can proceed when the agent has the requisite information. An agent may ignore any message or process messages in any order.

Given a protocol, Deserv enables implementing agents out of FaaS abstractions. The heart of Deserv is a generic adapter built via FaaS abstractions. Given a protocol and a role played by an agent, the adapter captures the necessary reasoning to guarantee compliance with the protocol: ensure validity of outgoing messages and verify validity of incoming messages. The adapter supports an interface for plugging in the agent's business logic, that is, its decision making. When the agents are realized in FaaS, we obtain the benefits of FaaS for a decentralized application. Specifically, Deserv modularizes an agent's decision making across as many FaaS functions as needed. Instances of a function are launched as needed by the serverless platform, thus taking advantage of its autoscaling capabilities.

To summarize, Deserv's contributions are the following.

- It is the first programming model geared toward decentralized applications in the cloud.
- The programming model is such that it naturally exploits FaaS features for concurrency and autoscaling.

The rest of the paper is organized as follows. Section II introduces to FaaS and introduces our running example. Section III introduces the idea of an information protocol. Section IV introduces a programming model for implementing serverless agents based on protocols. Section V gives details of the implementation of our programming model, especially how it leverages FaaS. Section VI presents a performance evaluation indicating that Deserv performs as well as traditional methods. Section VII concludes with a discussion of implications and future work. Section VIII provides links to our implementation.

II. FaaS

Existing FaaS programming models are based on stateless *functions*, also known as the *function-as-a-service* (FaaS) paradigm. In this paradigm, every function is a service that clients can request. Functions are supposed to be *stateless*, which is an architectural constraint that discourages storing session (client) state in the service [10]. Statelessness promotes scalability: A new service instance can be spawned for every request and can be garbage collected as soon as the request has been handled. Most FaaS implementations are stricter, imposing a maximum runtime for a service instance (15 minutes for AWS Lambda, after which the instance is killed). Most FaaS programming models support composing functions via workflows, i.e., as orchestrations.

Most interesting applications, however, are naturally stateful. Stateless functions can be used to implement stateful applications by persisting application state to a backend database. For example, a customer (as client) can place a purchase order (PO) with a merchant by invoking a function that returns an identifier for the PO. The PO would be persisted by the function in a database. Another function would enable the customer to retrieve the status of the PO. However, the stateless function programming model offers no abstractions for managing state. To address this limitation, some serverless

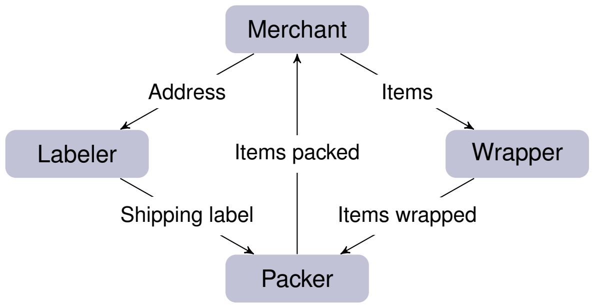

Fig. 1. Interactions in the PO Fulfillment scenario.

programming models have begun to offer stateful functions. For example, Microsoft Azure offers *Durable Functions* [11]. *Durable Entities*, an enhancement, map to actors [12], [13].

It is worth remarking that although a system of actors can be viewed as decentralized in the sense that the system is nothing more than actors communicating via messages, Deserv is concerned with abstractions that facilitate the representation and implementation of decentralized applications. Although actors could be used to implement agents, the notion of a protocol and a protocol-based programming model that facilitates implementing agents (our focus) are concerns orthogonal to the actor model itself.

Our running example is a simple scenario for purchase order (PO) fulfillment [4] that captures the essential elements of real-life decentralized applications. The scenario involves four parties: MERCHANT, WRAPPER, LABELER, and PACKER. MERCHANT receives POs, comprising one or more items, from a customer. MERCHANT requests WRAPPER to wrap each item and requests LABELER to create a shipping label with the customer's address. WRAPPER sends wrapped items to PACKER who puts items in shipping boxes (one per PO). LABELER sends a label for each PO to PACKER, who affixes it to that PO's box, and notifies MERCHANT for each item packed in the box. Figure 1 illustrates the information flow between the parties.

In FaaS, an application is either a function or, more generally, an orchestration of functions. For example, Amazon's AWS Step Functions enables composition of functions (and others services) into a workflow. An orchestration, however, is ill-suited for decentralized applications because it captures only one party's perspective. For example, MERCHANT, as orchestrator, may offer two functions as services, *POFunction* and *PackedFunction*. *POFunction* is triggered by incoming POs in response to which it sends requests to both LABELER and WRAPPER and stores its requests in a database. *PackedFunction* is triggered by notifications of packed items from PACKER; it checks that the notification corresponds to a previously logged request. The orchestration does not capture the entire decentralized application—it omits how LABELER and WRAPPER handle requests from MERCHANT, how they communicate with PACKER, and how PACKER proceeds.

III. Representing a Decentralized Application as an Information Protocol

Deserv begins from a protocol specifying a decentralized application. Specifically, we adopt *information protocols*, which specify information causality and integrity constraints [9]. Information causality captures information dependencies: what information must be known (and not known) in its *local state* (history of messages sent and received by it) to be able to send a message. Information integrity captures consistency in distributed settings: there cannot be two messages sent with conflicting information. Given the local state of an agent, an agent can send any message that satisfies the specified causality and integrity constraints.

We illustrate the main ideas of information protocols via Listing 1, which gives a protocol named *Fulfillment* that is enacted by roles MERCHANT, LABELER, WRAPPER, and PACKER. *Fulfillment* composes several message specifications, each with its sender and receiver roles, and some information parameters. For example, *RequestLabel* is from MERCHANT to LABELER and its parameters are orderID and address. A concrete message (instance) associates values (bindings) to the parameters, e.g., orderID to `10` and address to `123 Main Street`.

A. Information Integrity

Some parameters in a message specification are annotated *key*. Key parameters support integrity. A tuple of bindings for the key parameters of a message specification identifies an instance of the specification. Further, in an enactment, relative to the binding for the key parameters, a nonkey parameter may have at most one binding. This enables correct correlation across messages. For example, say *RequestLabel* occurs with bindings [orderID: `10`, address: `123 Main Street`]. Then, a *Labeled* that occurs with [orderID: `10`, address: `123 Main Street`, label: `GHT454`] would satisfy integrity. However, a *Labeled* with [orderID: `10`, address: `987 Elm Street`, label: `GHT454`] would violate integrity because for the same binding of orderID, there are different bindings of address.

Further, both *RequestWrapping* with [orderID: `10`, itemID: `X1`, item: `apple`] and *RequestWrapping* with [orderID: `10`, itemID: `X2`, item: `kiwi`] satisfy integrity since they feature different bindings for the composite key consisting of orderID and itemID. Assume the *RequestWrapping* with item `apple` has occurred. Then, *Wrapped* with [orderID: `10`, itemID: `X1`, item: `kiwi`, wrapping: `foil`] would violate integrity because for the same tuple of bindings for the composite key, there are different bindings of item.

Packed demonstrates a sophisticated correlation pattern where information with different keys is brought together: In any *Packed* instance, the binding for label must be correct with respect to the binding for orderID and the bindings for itemID and wrapping must be correct with respect to the bindings for orderID and itemID.

B. Information Causality

Every message parameter is adorned ⌜in⌝, ⌜out⌝, or ⌜nil⌝. Adornments capture information causality. For a parameter adorned ⌜in⌝, the sender must already know its binding from prior communications, that is, from its local state. For a parameter adorned ⌜out⌝, the sender must not already know its binding; however, in sending the message, the sender generates a binding for it (at which point the binding becomes known to the sender). For, a parameter adorned ⌜nil⌝, the sender must not already know its binding nor can it generate it (that is, the message is sent without any binding for the parameter). Note that we can only talk about knowing a parameter relative to some binding for the associated key. Thus, given some binding of orderID, we can talk about knowing or not knowing the binding of address; and relative to some tuple of bindings for the composite key comprised of orderID and itemID, we can talk about knowing or not knowing the binding of item.

We give some examples to make the idea concrete. All parameters of *RequestLabel* are adorned ⌜out⌝. Thus, MERCHANT can send a *RequestLabel* instance at any point (that is, in any local state) because it can generate bindings for them both. In *RequestWrapping*, orderID is adorned ⌜in⌝, whereas itemID and item are adorned ⌜out⌝. Thus, MERCHANT can send *RequestWrapping* by supplying an already known (binding of) orderID (from a previously sent *RequestLabel*) and generating bindings for itemID and item. Parameters orderID, itemID, wrapping, and label are all ⌜in⌝ in *Packed*; only status is ⌜out⌝. Thus, MERCHANT can send *RequestWrapping* by supplying already known orderID, itemID, wrapping, and label (from received Wrapped and Labeled messages) and generating a binding for status.

Notice that the only way to generate a parameter binding is via an ⌜out⌝ adornment. And once a binding has been generated, there is no way to update it. That is, information generated in a protocol enactment is *immutable*.

C. Transport Assumptions

An information protocol constrains only the emission of messages by agents. The protocol itself imposes no constraints upon message reception; specifically, a message can be received by an agent at any point, that is, no matter what its local state. This means that no ordered delivery transport or middleware, such as TCP or message queues, is required for protocol enactment; the protocol can be enacted over an unordered transport such as UDP. Further, once a message has been recorded in an agent's local state, it can be retransmitted as often as needed because receiving a message more than once makes no difference to the receiver's local state. Retransmissions make it possible to enact the protocol reliably over lossy transports such as UDP [4], [14].

D. Expressiveness

Although *Fulfillment* does not feature any mutually exclusive choice between messages, information protocols can express choice by setting up conflicts between messages. For example, imagine that WRAPPER has a choice between

Listing 1. The *Fulfillment* Protocol

```
Fulfillment {
 role Merchant, Wrapper, Labeler, Packer

 parameter out orderID key, out itemID key, out item, out status

 Merchant-> Labeler: RequestLabel[out orderID key, out address]
 Merchant-> Wrapper: RequestWrapping[in orderID key, out itemID key, out item]

 Wrapper-> Packer: Wrapped[in orderID key, in itemID key, in item, out wrapping]
 Labeler-> Packer: Labeled[in orderID key, in address, out label]

 Packer-> Merchant: Packed[in orderID key, in itemID key, in wrapping, in label, out status]
}
```

sending *Wrapped* and *Decline* (to MERCHANT). A protocol could express that by introducing the same parameter, say decision in both messages and adorning it ⌜out⌝ in both. Now if WRAPPER sends one of them, decision becomes bound, which makes it impossible to send the other.

Information protocols being declarative don't employ traditional control-flow constructs such as sequencing and loops. The causality constraints achieve the effect of sequencing and the notion of keys enables repetition in an interaction. For example, *Fulfillment* enables the agents to handle as many POs as needed (each identified by orderID), each with as many items (each identified by itemID) as needed.

Information protocols have been extended to support dynamic role binding, roles being played by multiple agents, and multicast [15]. In this paper, however, we confine ourselves to the basic representation as illustrated above.

E. Threats to Integrity

In the foregoing choice example, the choice between *Wrapped* and *Decline* is local to the WRAPPER. Therefore, WRAPPER can ensure that if one happens, then the other doesn't. However, it is possible to write a protocol where the same parameter may be bound concurrently by multiple agents. When enacting such a protocol, local checking by agents in insufficient to ensure integrity. Such a protocol is *unsafe*. It can be verified statically where a protocol is safe or not [16]. An unsafe protocol is not fit as a basis for application design.

If a protocol were safe and agents were compliant in emitting messages, then integrity would be guaranteed under the (routinely-made and practical) assumption that the underlying transport does not deliver corrupt messages. To protect itself from noncompliant (misbehaving) agents, an agent can defensively check received messages for integrity before inserting them into their local state [17]. Agents designed following Deserv are guaranteed to be compliant to the protocol. If all agents enacting a protocol were (correctly) implemented following Deserv, then such defensive checking wouldn't be necessary, but it is good practice to check.

F. Completion and Composition

Notice the parameter declaration line toward the beginning of *Fulfillment*. It lifts the notion of instance to protocols. The line means that each tuple of bindings for orderID, itemID, item, and status constitutes a *complete* instance of the protocol. A protocol is *live* if any instance can progress to completion [16].

The parameter line supports protocol composition by describing an "interface" for the protocol, saying what parameters the protocol generates via its own computations (those adorned ⌜out⌝) and what parameters needs from other protocols (those adorned ⌜in⌝). In *Fulfillment*, all parameters are adorned ⌜out⌝ in the parameter line, which means that it can be enacted *standalone*—without composing with other protocols.

Our tool suite (available at https://gitlab.com/masr/protocheck) checks protocols for liveness, safety, and other properties.

G. Public versus Internal Computation

Protocol enactment captures the *public* computation in a decentralized application. A protocol leaves an agent's *internal* computation, as reflected in its decision making, unspecified. The internal computation determines whether the agent sends a message and the bindings of the ⌜out⌝ parameters in the message. For example, WRAPPER could determine by some internal business logic that a particular *Wrapped* message should be sent with wrapping bound to `silk` and not `plastic`. Note though that it is only the sending of the message (specifically, the recording of the message in the local state) that generates the binding in the public computation.

IV. PROGRAMMING MODEL

The Deserv programming model shows how to realize decentralized applications on FaaS. We specifically used AWS Lambda (whose functions are referred to as lambdas). Specifically, given a protocol, the programming model enables separately constructing each agent that plays a role in the protocol. Each agent is guaranteed to be compliant with the protocol. A working implementation of the adapter is available along with the rest of our code at https://gitlab.com/masr/deserv.

A. Adapter

The Deserv programming model is realized in a generic protocol adapter that resides in each agent. Given a protocol and the role the agent plays in it, the adapter ensures that the agent only sends and receives compliant messages. Thereby, Deserv standardizes and facilitates the implementation of any agent playing a role in any protocol.

The adapter captures the crux of decentralization: each agent has *local* knowledge of each protocol enactment [17]. An agent's local state comprises precisely the information that it sends or receives in a message: hence, the local state is shared with others. In addition, each agent maintains *internal* state to support its decision making. Internal state cannot be shared except by copying parts of it to a message, i.e., adding it to the local state.

Figure 2 shows the components of an adapter and how it interfaces with the rest of the agent. Filled boxes are functional components. The Receiver and Emitter interface with the communication infrastructure to transmit messages. The Checker maintains the local state. For an outgoing message, it verifies causality and integrity (the local state contains bindings for all ⌜in⌝ parameters and for no ⌜out⌝ and ⌜nil⌝ parameters) and updates the state by inserting bindings for the ⌜out⌝ parameters. For an incoming message, it verifies integrity: that there isn't already a conflicting binding for any ⌜in⌝ or ⌜out⌝ parameter. The Checker logs and discards incoming or outgoing messages that fail its checks.

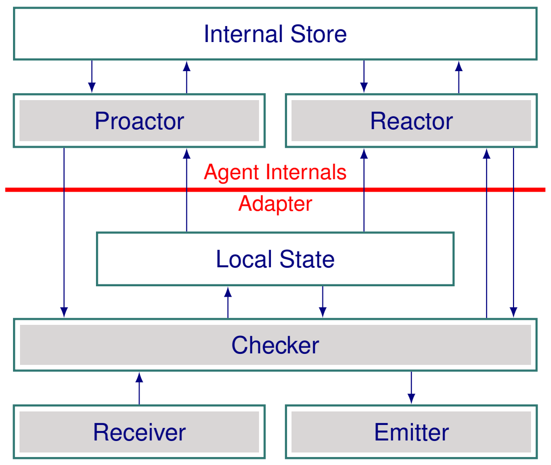

Fig. 2. Agent architecture schematically, showing the components involved in the agent internals and the generic adapter and interfaces between the components. Functional components (which map to FaaS functions) are filled gray boxes, state components are not filled.

In our reference implementation, each agent is a composition of multiple functions, specified in a serverless framework configuration file that lists the resources (e.g., state databases) and functions, and specifies the events they listen for and how they are to be invoked. Within the agent, the functional components (represented in the diagram by filled boxes) are each implemented as FaaS functions. The Emitter, Receiver, and each Reactor and Proactor are implemented as separate FaaS functions that can be scaled up to multiple instances. By default, the Emitter and Receiver communicate using JSON objects transmitted over HTTP, but in theory any communication infrastructure supported by AWS may be used. Communication between the components of an agent is done via direct, asynchronous lambda invocations. To ensure consistency, we limit the Checker to one instance per agent; a more sophisticated implementation might use synchronization primitives such as locks or sharding based on enactment keys instead.

The Local State is implemented using DynamoDB, which is simply used to store message objects indexed by their enactment keys. By logging every message in the Local State table, and using it as the official agent history during compliance checks, the agent can easily be resumed after a crash or lambda timeout. The Internal Store is more abstract, referring to any other knowledge or components used by the agent implementation in its decision making if necessary; perhaps other functions or databases. For example, in our logistics scenario, the MERCHANT uses a DynamoDB table of purchase orders to trigger enactment of the protocol.

This design has several implications. First, it is somewhat scalable, because the individual components and logic of the agent are run concurrently, and some of them can themselves be replicated. The main limitations for scaling are the Checker and Local State, which must operate synchronously to ensure there are no race conditions between message events. Other than these points, fine-grained synchronization is unnecessary, because the agents operate asynchronously and independently of each other, and the protocol they enact (if verified safe) ensure that no inconsistency can arise between agents. The complexity of using multiple FaaS functions to represent a single agent does make debugging somewhat harder should they break, but most of the components are simple, and the main agent behavior is funneled through the core adapter component (Checker), meaning that during normal operation the Checker's log is the only one that need be examined.

B. Agent Reasoning

An agent may have several instances of the two decision-making components, Proactor and Reactor. Proactors and Reactors are implemented as separate FaaS functions, and have separate entries in the agent's serverless framework configuration file.

Proactors are not invoked by the adapter, but can be configured to receive other events. A Proactor is proactive and can initiate actions independently of other events; that is, a Proactor can send messages that are not directly in response to other messages. In a self-contained application, at least one agent must include a Proactor to initiate the enactment. MERCHANT initiates the logistics scenario by proactively sending *RequestLabel*. In our implementation, MERCHANT sends *RequestLabel* in response to a database event (when a PO is submitted), but that is not a protocol event, and so not automatically handled by the Checker.

Reactors are invoked by the adapter according to a configuration file mapping message schemas to FaaS functions. Each Reactor handles one message schema, communicated as a function invocation passing a JSON object encoded in a string as the payload. For example, when LABELER receives a *RequestLabel* message from MERCHANT, it invokes the corresponding Reactor with the contents of the message and some relevant enactment history information. The Reactor may then generate a label, which it passes on to LABELER's adapter for checking and logging before sending it to PACKER.

C. Deployment

Deployment is managed by serverless framework tools. A single configuration file describes all the components of an agent, as well as any resources it might need. Within that composition specification, the developer references further "layers" (composable filesystem data) containing files that configure the individual components. One file contains the protocol, encoded in JSON for the adapter's use, primarily for checking schemas and identifying which role should receive which messages. Another file maps the role names to the URLs of their Receiver endpoints, so the adapter knows how to contact the recipients of its messages.

V. IMPLEMENTATION

The previous section described our programming model and architecture at a high level, which should be applicable to any FaaS platform. However, for greater clarity, we now present some details about our reference implementation.

We used the Serverless Framework tooling for configuration and deployment, but ultimately used many AWS-specific components, so our implementation is not cross-platform. Our logistics scenario implementation is split into two main pieces, the shared adapter components and the scenario-specific agent implementations.

An important concept from cloud deployments is the *stack*, which is a collection of resources managed together. AWS has many distinct services, each with their own unique configuration parameters; to simplify the deployment of a system which might require multiple services (e.g., API gateway, Lambda functions, and databases), AWS provides a configuration management tool called CloudFormation, which uses declarative specifications to deploy multiple related resources at the same time and interconnect them. The Serverless Framework manages the configuration of multiple resources in a cross-platform way, using the name *service*.

Below, we give snippets from Serverless Framework service files to illustrate how to configure and deploy a Deserv system, but we adopt the AWS term (stack) for clearer distinction from web services. We can deploy the stack by passing it to the cloud deployment system (e.g., CloudFormation), or the stack specification may be interpreted locally by the Serverless Framework tools to instantiate all the component resources. Since the stack is a declarative specification, CloudFormation or the framework tools adjust any settings to match the new desired values, and atomically deploy or rollback the entire stack.

We use a separate stack for each agent, because they are separate compositions of multiple functions and resources that should not be tightly coupled.

A. Adapter Components

As described in Section IV, our adapter consists of three main components: Receiver, Emitter, and Checker. In our implementation, these three are bundled together into a component library, which can be deployed as its own reusable stack, so that it only needs to be uploaded once and can then be referenced by each agent.

Listing 2. Checker component layer specification

```
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, a string specifying
       license information
  package:
    include:
      - '!./**'
      - checker.py
```

Listing 2 shows a snippet from the component stack's serverless configuration that specifies the Checker component. This snippet declares the *layer* for the checker component, and declares it to contain the `checker.py` file. Other parts of the configuration file export this layer with a public name so that the agents can refer to it, or declare basics about the components and their dependencies so they can be deployed in a local test environment.

These components need to be deployed to AWS to be available resources for deploying agents. Separating these common components into a separate stack makes developing them somewhat more complex, because they must be separately redeployed when changes are made. Once they are stable and deployed, however, they form a reusable resource that can be referenced by every subsequent agent instead of being replicated.

B. Scenario Components

Once the adapter components have been deployed, the scenario needs to be implemented and deployed. Because each agent is itself a composition, each agent has its own serverless framework configuration file and must be separately deployed. Since real-world decentralized systems would have interacting agents from multiple organizations, no single organization would be responsible for deploying all the agents. Therefore, separate deployment of agents is preferable. However, to simplify this process somewhat, we did write a script for applying a serverless command (e.g., deploy or remove) to all components necessary for the scenario.

1) Scenario Configuration: A multiagent system is implemented according to one or more protocols that are enacted by its constituent agents. When two or more agents in the same system are deployed by a single organization, it may be helpful to deploy the configuration as a separate stack so that it can be referenced instead of copied by each of those agents. Unfortunately, referencing another stack does *not* enable dynamic propagation of updates to the configuration; each agent must be redeployed if the configuration changes, because the agents import a specific instance of the configuration layer rather than link to an abstract dynamic reference.

Listing 3. Role Endpoint Mapping File

```
{
  "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"
}
```

The configuration layer in our example scenario implementation contains two files: the role endpoint mapping file, and a JSON representation of the protocol.

Listing 3 shows an example endpoint mapping file, mapping each role name to the HTTP endpoint of an agent's Receiver. These endpoints are invoked using a POST request to submit a JSON object encoding the message contents, which the Receiver parses and hands off to the Checker.

Listing 4 gives the first portion of the *Logistics* protocol represented in JSON, for use by the Checker. The primary use of this specification is to identify each message's schema and recipient for compliance checking and routing, respectively.

2) Agent Specification: To illustrate how scenario components are specified, we examine MERCHANT because it includes both a Reactor and a Proactor.

First, MERCHANT's serverless configuration file specifies all the components it needs—not only its Proactor and Reactor, but also its copies of Emitter, Receiver, and Checker.

Listing 4. JSON Protocol Specification

```
{
  "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"
    },
    ...
```

Listing 5 shows the portion of MERCHANT's configuration that specifies the Checker component. It declares a new function `MerchantChecker`, which uses the `CheckerLayerExport` to load the checker code, which includes the Checker, and loads the configuration layer described above. Uniquely, the Checker component is a singleton; all other components may freely scale.

Listing 5. MERCHANT 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
```

Listing 6 shows how a Reactor is declared. It is given a name, its lambda handler is identified, and the necessary code is loaded.

Listing 6. MERCHANT's Reactor for *Packed*

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

In this case, all the Reactor does is log successful processing of the item, as shown in Listing 7.

Listing 7. MERCHANT's Reactor for Packed

```
def lambda_handler(event, context):
    print("Reactor of Packed message: " +
          str(event["message"]) + "; Enactment is " +
               str(event["enactment"]))
```

To register the Reactor, another configuration file is provided to map the message schema names to their handling Reactor functions. The configuration snippet in Listing 8 shows how the `reactors.json` file is added for MERCHANT; the package specification applies to all the functions, indicating that they should exclude all files, but include `reactors.json`.

Listing 8. MERCHANT Package Specification

```
package:
  individually: true
  include:
    - '!**'
    - reactors.json
```

MERCHANT's `reactors.json` file, shown in Listing 9, maps the names of messages the agent can receive to the Amazon Resource Name (ARN) of the function that should be invoked to handle it. These ARNs don't exist until the function has been deployed once, so the Reactors must be deployed and added to this mapping file before the Checker is deployed the first time.

Listing 10 gives a slightly more complex Reactor, implementing PACKER's reaction to the *Wrapped* message. This example shows more clearly how the enactment context can be used to correlate information from multiple messages to reason about. Proactors differ from Reactors in that they are not invoked by the Checker in response to a received message, and must be triggered by some other event.

Listing 11 shows the portion of MERCHANT's serverless framework function specification that declares its Proactor. The details are likely unique to this Proactor, but give an example of how Proactors can work. For this scenario, we give MERCHANT an extra POST endpoint for customers to submit their purchases, which populates a database. Proactors differ

Listing 9. MERCHANT Reactor Mapping

```
{
  "Logistics/Packed": "arn:aws:lambda:us-east-1:834106683512:function:Merchant_Packed_Reactor"
}
```

from Reactors in that they are not invoked by the Checker in response to a received message, so they must be triggered by some other event. The `ordersTable` generates events when POs are added, triggering `PO_proactor`, which initiates a new enactment of *Logistics*.

Listing 10. PACKER's Reactor for *Wrapped*

```
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",
            InvocationType="Event",
            LogType="Tail",
            Payload=payload,
        )
        print(response)
```

Listing 11. MERCHANT Proactor Specification

```
# functions
order:
  handler: order.writeToDynamo
  events:
    - httpApi: POST /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
```

VI. EVALUATION

A Deserv agent is modular. It encapsulates reasoning about interactions separately from its internal reasoning encoded in its Proactors and Reactors.

Contrast a Deserv agent for MERCHANT with the orchestrator agent described in Section II. Suppose both agents implement the same functionality—i.e., the orchestrator is programmed to perform the same checks as the Deserv agent.

The Deserv agent presents greater opportunities for concurrency (involving its adapter, a Proactor for processing POs, a Reactor for incoming Packed messages, the emitter, and receiver) than the orchestrator (which comprises just two functions).

Below, we present results from an empirical study that demonstrates that a Deserv agent may be executing several instances of the each of its components at the same time (except the Checker, of which there can be only one instance at any time to ensure local state consistency). We will refer to the deployed instance of the Merchant agent as M-Agent to distinguish it from the conceptual Merchant role.

A. Experimental Design

We conducted an experiment deploying *Fulfillment* on AWS Lambda to verify Deserv's performance and scalability. For each run, we randomly generated 1,000 POs, each containing one to four items, using a script in our repository that would asynchronously submit them to M-Agent via HTTP. We set all DynamoDB tables to autoscale, and did not throttle any requests. We submitted POs via HTTP request to another Lambda function that stored them in a separate PO table. M-Agent's Proactor subscribed to this table's update event stream and initiated the enactments.

We considered two settings: *normal*, and *delayed*, with a one-second delay to each Reactor and Proactor—to simulate heavier processing than the adapter. The delay was implemented by adding a sleep statement to the beginning of each Reactor's lambda_handler to increase its execution time by one second.

Performance results were computed by exporting M-Agent's history table and analyzing the message timestamps using a script included in our repository. We also examined the maximum number of concurrently running instances of the Lambda functions during an experiment using the online AWS CloudWatch monitoring console.

B. Results

Table I shows our performance results. Both settings yield similar average duration—low effect size (Cohen's d = 0.015)—and throughput. Reactor computation without delay ranged from 1ms to 380ms, so a delay of 1s is substantial. Yet, average processing duration and throughput change only slightly even with the added delay.

TABLE I
PERFORMANCE RESULTS

	PO Duration		Throughput	
Experiment	Mean (s)	St. Dev. (s)	POs/s	Items/s
Normal	266.51	51.45	1.23	2.37
Delayed	267.27	46.45	1.21	2.34

Although it is possible that the added delay was insufficient to outweigh the Checker's bottleneck, we can infer that AWS Lambda scaling was able to compensate for some of the increased Reactor duration by looking at the maximum concurrency of each of M-Agent's components. In the normal setting, we observe a maximum of three concurrent emitter instances, two receiver instances, and two Reactor instances. With delay, we observe up to five emitters and two receivers, but 13 Reactor instances, since each Reactor takes much longer to handle a single message. This indicates that Deserv benefits from FaaS's flexible scaling to maintain throughput.

Although our setting is much smaller than the thousands of concurrent functions that AWS Lambda is capable of, the results demonstrate that the Deserv architecture can take advantage of automatic function scaling. The singleton Checker component and database interactions may be a bottleneck, but the Reactors and Proactors are where the business logic is processed. If one of the Reactors happens to take a long time to process a message, up to a 15-minute execution limit for AWS Lambda functions, then many instances could be run concurrently to handle the same throughput as the Checker and Local Store.

VII. Discussion

We conceptualized a decentralized application as consisting of agents who represent autonomous principals and interact via asynchronous messaging. Any decentralized setting necessarily motivates a protocol by which the agents communicate. The architectural choice is between specifying and not specifying the protocol. Specifying the protocol makes it possible to build loosely coupled applications: Each agent can be constructed and maintained independently based on the protocol. Current approaches don't specify the protocol formally, as a consequence of which the resulting application is either realized in a centralized fashion, e.g., as an orchestration, or in an operationally decentralized but tightly coupled way. Deserv can be used wherever a loosely coupled architecture is desired even if no autonomous principals are involved.

Deserv is a protocol-based programming model for building decentralized serverless applications. It enables building compliant agents in a modular fashion via abstractions such as the Proactor and Reactor, which developers use for plugging in the agent's decision making. In fact, Deserv shares serverless computing's ambition of letting developers focus on the business logic, but goes farther in that an agent developer need not worry about issues that messaging in a decentralized setting pose. For example, the agent developer need not handle message orders or correlation and need not write code to interface with the communication infrastructure itself. In a sense, the developer programs to the adapter.

Deserv is modular at two levels. First, it separates the agents realizing a decentralized application from one another by reducing their coupling to what's specified in a protocol—precisely what's needed for the application. Second, it structures each agent's internals so that an agent becomes a composition of microservices, ideally suited for FaaS. Information in protocol enactments is immutable, which improves concurrency within and between agents. Immutability is well-aligned with functional programming, which is a promising connection to explore further.

Deserv simplifies programming decentralized applications because it adopts information protocols. This approach is declarative and contrasts with traditional approaches for specifying interactions between autonomous principals as a *choreography*, which specifies a control flow of messages [18], [19]. Choreographies, however, do not support flexible interactions as they require the enacting parties to move in lockstep. Specifically, in a choreography, if a party sends a message to another, the latter must be in a state where it can receive the message. Further, choreographies typically assume ordering guarantees (e.g., FIFO) from the communication infrastructure. Information protocols address these and other limitations of choreographies [20].

The Financial products Markup Language (FpML) [21] exemplifies current approaches to realizing decentralized applications. Message formats are described in enormous detail; however, the business protocols themselves are specified via UML interaction diagrams, an informal notation. This means that there is little support by way of a programming model for implementing agents. Specifying financial interactions as information protocols would enable using Deserv for programming agents. Deserv enables programming based on the structure of interactions; it assumes low-level operational details such as message formats. We anticipate that applying Deserv to model and implement financial interactions will reveal the need for practical enhancements to Deserv.

Abstractions for building stateful serverless applications is a direction in serverless computing. Ongoing work in this direction is targeted toward unitary applications that coordinate via shared memory [22]–[25]. Deserv is unique in that it addresses the modeling and implementation of *stateful* decentralized applications in the serverless paradigm. Importantly, agents do not share state. The application state may be viewed as a vector, each of whose elements is the local state of one agent; however, the application state is not materialized anywhere. Each agent's local state is managed by its Checker that ensures it is updated only with correct messages, as described in Section IV. Constraining updates in this manner ensures consistency even when the agents send messages in true concurrency over an asynchronous infrastructure that doesn't preserve message order.

The components internal to an agent, as shown in Figure 2, share the agent's local state; however, the Checker is the only module that can update the local state and to avoid inconsistencies, it serializes updates requested by the other components. This serialization is a bottleneck and techniques for alleviating it would be a worthwhile direction of research. Traditional systems techniques could be complemented with information-based techniques. For example, we could exploit the fact that key bindings identify unrelated protocol instances, which could be updated concurrently.

Deserv has some similarities with other declarative service and composition specification frameworks such as OpenAPI [26] and CloudFormation [27]. OpenAPI declaratively specifies the API of a service as the endpoints and schemas it supports; In contrast, BSPL specifies an interaction—all the messages and dependencies between multiple agents. Also, BSPL agents may send messages proactively, not just respond to API calls. AsyncAPI [28] extends OpenAPI to support event-based systems, with synchronous pub/sub channels, rather than synchronous request/response endpoints alone. AsyncAPI is closer to BSPL, but it does not describe the message dependencies, only their schemas. CloudFormation is the AWS solution for declaratively specifying and deploying compositions of cloud resources within the *same* context; Deserv enables interactions between agents *across* organizational boundaries. Deserv and CloudFormation are complimentary; in fact, Deserv uses CloudFormation to deploy agent components as described in Section V.

Current approaches for building distributed applications rely on special fault handling transport or middleware. However, doing so violates the end-to-end principle [29], according to which faults are best handled at the application level because the relevant information is only available at the application level. In fact, what constitutes a fault and how to handle it are both application-level considerations. Incorporating protocol-based abstractions for fault tolerance [4], [14] into Deserv is an important future direction.

VIII. Reproducibility

Our toolchain for verifying the correctness of BSPL protocols is available at https://gitlab.com/masr/protocheck. An implementation of Deserv is available at https://gitlab.com/masr/deserv. The `distributed` directory in the Deserv repository contains the adapter components, a deployment script, and a sample implementation of the *Logistics* scenario with a `run_test.py` script for submitting POs, and a `process_results.py` script for analyzing the M-Agent's history dump.

Acknowledgments

Christie and Chopra were supported by EPSRC grant EP/N027965/1 (Turtles). Singh was supported by the National Science Foundation under grant IIS-1908374.

References

[1] A. K. Chopra and M. P. Singh, "From social machines to social protocols: Software engineering foundations for sociotechnical systems," in *Proceedings of the 25th International World Wide Web Conference*. Montréal: ACM, 2016, pp. 903–914.

[2] N. Desai, A. U. Mallya, A. K. Chopra, and M. P. Singh, "Interaction protocols as design abstractions for business processes," *IEEE Transactions on Software Engineering*, vol. 31, no. 12, pp. 1015–1027, December 2005.

[3] A. Günay and A. K. Chopra, "Stellar: A programming model for developing protocol-compliant agents," in *Preproceedings of the 6th International Workshop on Engineering Multi-Agent Systems*, ser. LNCS, vol. 11375. Stockholm: Springer, 2018, pp. 117–136.

[4] S. H. Christie V, D. Smirnova, A. K. Chopra, and M. P. Singh, "Protocols over Things: A decentralized programming model for the Internet of Things," *IEEE Computer*, vol. 53, no. 12, pp. 60–68, 2020.

[5] M. Roberts, "Serverless architectures," https://www.martinfowler.com/articles/serverless.html, May 2018.

[6] A. Jangda, D. Pinckney, Y. Brun, and A. Guha, "Formal foundations of serverless computing," *Proceedings of the ACM on Programming Languages (OOPSLA)*, vol. 3, pp. 149:1–149:26, Oct. 2019.

[7] M. P. Singh and M. N. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*. Chichester, United Kingdom: John Wiley & Sons, 2005.

[8] J. Lewis and M. Fowler, "Microservices," https://www.martinfowler.com/articles/microservices.html, Mar. 2014.

[9] M. P. Singh, "Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language," in *Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. Taipei: IFAAMAS, May 2011, pp. 491–498.

[10] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[11] Microsoft, "Azure durable functions documentation," https://docs.microsoft.com/en-us/azure/azure-functions/durable/, accessed: 7 Jun 2021.

[12] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial Intelligence*, vol. 8, no. 3, pp. 323–364, Jun. 1977.

[13] G. A. Agha, *Actors*. Cambridge, Massachusetts: MIT Press, 1986.

[14] S. H. Christie V, A. K. Chopra, and M. P. Singh, "Bungie: Improving fault tolerance via extensible application-level protocols," *IEEE Computer*, vol. 54, no. 5, pp. 44–53, May 2021.

[15] A. K. Chopra, S. H. Christie V, and M. P. Singh, "Splee: A declarative information-based language for multiagent interaction protocols," in *Proceedings of the 16th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. São Paolo: IFAAMAS, May 2016.

[16] M. P. Singh, "Semantics and verification of information-based protocols," in *Proceedings of the 11th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*. Valencia, Spain: IFAAMAS, Jun. 2012, pp. 1149–1156.

[17] ——, "LoST: Local State Transfer—An architectural style for the distributed enactment of business protocols," in *Proceedings of the 9th IEEE International Conference on Web Services (ICWS)*. Washington, DC: IEEE Computer Society, Jul. 2011, pp. 57–64.

[18] X. Fu, T. Bultan, and J. Su, "Conversation protocols: A formalism for specification and verification of reactive electronic services," *Theoretical Computer Science*, vol. 328, no. 1-2, pp. 19–37, 2004.

[19] WS-CDL, "Web services choreography description language version 1.0," Nov. 2005, www.w3.org/TR/ws-cdl-10/.

[20] A. K. Chopra, S. H. Christie V, and M. P. Singh, "An evaluation of communication protocol languages for engineering multiagent systems," *Journal of Artificial Intelligence Research*, vol. 69, pp. 1351–1393, 2020.

[21] FpML, "FpML 5.11 recommendation," https://www.fpml.org/spec/fpml-5-11-9-rec-1/, Dec. 2019, accessed: 8 Jun 2021.

[22] P. Leitner, E. Wittern, J. Spillner, and W. Hummer, "A mixed-method empirical study of Function-as-a-Service software development in industrial practice," *PeerJ Preprints*, vol. 6, p. e27005, 2018.

[23] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the FaaS track: Building stateful distributed applications with serverless architectures," in *Proceedings of the 20th International Middleware Conference*. New York, NY, USA: ACM, 2019, pp. 41—54.

[24] S. Shillaker and P. R. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proceedings of the 2020 USENIX Annual Technical Conference*, 2020, pp. 419–433.

[25] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful Functions-as-a-Service," *Proceedings of the VLDB Endowment*, vol. 13, no. 11, pp. 2438–2452, 2020.

[26] OpenAPI, "OpenAPI specification v3.1.0," 2021, the Linux Foundation. [Online]. Available: https://spec.openapis.org/oas/v3.1.0

[27] AWS, "CloudFormation," 2021, Amazon AWS. [Online]. Available: https://aws.amazon.com/cloudformation/

[28] AsyncAPI, "AsyncAPI specification 2.0.0," 2019, AsyncAPI Initiative. [Online]. Available: https://www.asyncapi.com/docs/specifications/v2.0.0

[29] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–288, Nov. 1984.