

Protocols Over Things: A Decentralized Programming Model for the Internet of Things

Samuel H. Christie V, North Carolina State University and Lancaster University

Daria Smirnova and Amit K. Chopra, Lancaster University

Munindar P. Singh, North Carolina State University

Current programming models for developing Internet of Things (IoT) applications are logically centralized and ill-suited for most IoT applications. We contribute Protocols over Things, a decentralized programming model that represents an IoT application via a protocol between the parties involved and provides improved performance over network-level delivery guarantees.

The Internet of Things (IoT) enables new applications that leverage the capabilities of things—Internet-accessible devices that sense or control their environment. Many such applications involve autonomous parties who share information from

and exercise control over things to effectively cooperate in achieving their respective goals. For example, a patient could share information with a health care provider to receive automatically dispensed medication. Patients should have control over their health information and doctors over how they prescribe medications. More mundane IoT applications exhibit multiple parties too. For example, a smart home could involve an owner, one or

Digital Object Identifier 10.1109/MC.2020.3023887
Date of current version: 19 November 2020

more tenants, a power utility, and a security service. Likewise, in an enterprise, different suborganizations have different responsibilities and powers. That is, real-life applications involve multiple logical loci of control.

However, current programming models are geared toward a single locus of control: they violate the autonomy of all but one of the parties and don't jibe with things often being small and numerous. For instance, popular frameworks such as Node-RED (<https://nodered.org/>) and Eclipse Kura (<https://www.eclipse.org/kura/>) model an IoT application as an orchestration⁵ that receives information from sensors, processes it in a workflow, and effects actions in the environment.

In contrast, we advocate a programming model called *Protocols over Things (PoT)* that is not only distributed physically but also decentralized.¹¹ PoT reflects everyone's autonomy and decouples their reasoning through asynchronous communication. Specifically, PoT captures a decentralized application via a protocol that specifies the communication constraints between the parties, abstracted as roles. PoT can employ IoT communication standards such as MQTT,⁴ the Constrained Application Protocol,⁸ and the User Datagram Protocol (UDP).⁶

Using a logistics scenario that we implement in Node-RED according to PoT, in this article we demonstrate how a protocol specification enables the use of code generation tools and a generic communication adapter to simplify development. We focus specifically on how the adapter correlates messages according to the protocol and compare PoT to existing approaches for message correlation in Node-RED. We further demonstrate how PoT embodies the end-to-end principle⁷ by

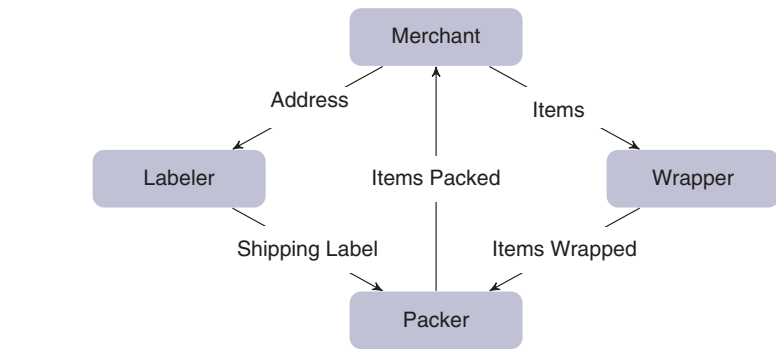


FIGURE 1. A conceptual model of the logistics scenario relative to a single purchase order.

supporting deployment over unreliable asynchronous communication mechanisms instead of relying on transport-layer delivery guarantees. Finally, we experimentally demonstrate that a PoT implementation over UDP with application-level retry

purchase order (PO)—imagine an external customer. Each PO includes one or more items and a shipping address. Merchant sends the address to Labeler, which generates the appropriate shipment label to be affixed to the shipping box. Labeler sends the generated label

CURRENT PROGRAMMING MODELS ARE GEARED TOWARD A SINGLE LOCUS OF CONTROL.

strategies compares favorably with the same implementation over MQTT.

EXAMPLE IoT SCENARIO: LOGISTICS

We adopted a warehouse logistics scenario,⁹ simplified to focus on the aspects relevant to decentralization: the parties, their communications, and their decision making.

Figure 1 illustrates this scenario conceptually. Here, Merchant, Wrapper, Labeler, and Packer are autonomous parties; the arrows indicate information flows. Merchant receives a

to Packer. Merchant sends information about the items to the Wrapper, which wraps the items appropriately for shipping (for example, paper for durable items and bubble wrap for fragile ones) and notifies Packer they are ready. Packer affixes the shipment label to a box and notifies Merchant for each item (in the PO) that it packs in the box.

Each party applies its decision making in deciding when and what information to communicate. For example, Wrapper may select a wrapping based on current inventory and cost, and Labeler may select a shipping label

NODE-RED

Node-RED is an interactive programming and execution environment for IoT applications. A web interface presents a palette of function blocks called nodes. A user can construct a flow—technically, an orchestration—by connecting nodes via virtual wires that run from an output port of one node to the input port of another. A flow typically starts with one or more nodes that sense information from the environment and ends with some action in the environment.

Figure S1 illustrates a flow. The type of each node is indicated by its color and icon. Each node has an informal name.

In Figure S1, the nodes named Labeled and Wrapped are MQTT subscriptions to the topics Labeled and Wrapped, respectively. The output ports of both Labeled and Wrapped are connected to the input port of the Join node, meaning that

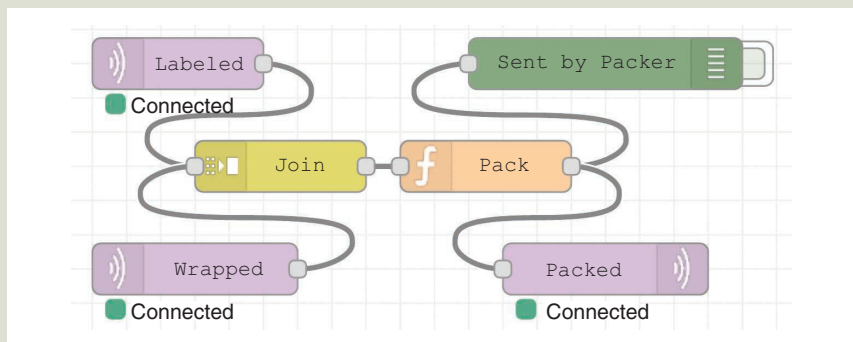


FIGURE S1. A simple Node-RED flow using Join to combine messages.

received messages are passed onto the Join node. This node aggregates multiple messages together (in this example, one message from each of the MQTT topics). Pack is a function node that applies custom JavaScript code to the aggregates produced by Join. The output port of Pack is connected to two other nodes, which means that each of its outputs is copied and passed to those nodes separately. The node named Sent by Packer logs its inputs to the developer console to facilitate debugging. The node named Packed is an MQTT node that publishes any input it receives to the Packed topic.

based on the speed and cost of the shipper. A party may choose to delay or not send a message; for example, Wrapper may hold on to an item until it has the appropriate wrapping, and Labeler may discard invalid addresses. In general, the parties work concurrently and asynchronously based on information available to them.

CHALLENGES IN PROGRAMMING IoT APPLICATIONS AS ORCHESTRATIONS

Node-RED (see “Node-RED”) epitomizes the orchestration approach to programming IoT applications. How might one implement a decentralized IoT application using flows? Each

party’s computations may be captured in a separate flow called the party’s endpoint that communicates with the endpoints of other parties via messages over, for example, MQTT.

Figure 2 illustrates the resulting architecture schematically. Notably, there is no representation of the application that captures its decentralized

nature except as endpoints implemented in Node-RED.

There is another shortcoming in the architecture in Figure 2. To interoperate, the parties need a specification of the structures of the messages they communicate. Such a specification lies strictly outside Node-RED but is crucial to decentralization. Lacking such a specification, the endpoints would become tightly coupled.

Suppose such a specification is available, as is common in practice. For example, Listing 1 specifies the structures of the messages alluded to in Figure 1. Sadly, the message structures in Listing 1 are inadequate for capturing the scenario, particularly its correlation requirements. Packer needs application-level knowledge to correlate the items and shipping label for the same PO and pack them correctly, since it receives them separately.

Since Node-RED, as a general purpose platform, provides only low-level programming facilities unaware of the application semantics, an implementation of correlation using standard Node-RED features will, at best, be ad hoc and difficult to maintain. Below, we describe three increasingly sophisticated approaches for implementing the correlation necessary for Packer using standard Node-RED facilities, to explain their advantages and limitations. Working Node-RED flows implementing the logistics scenario with each of the approaches are available along with the rest of our code at <https://gitlab.com/masr>.

Join based

Figure S1, in fact, depicts an endpoint for Packer. Recall that Join in the figure produces an aggregate consisting of one message from each of the labeled and wrapped MQTT subscriptions.

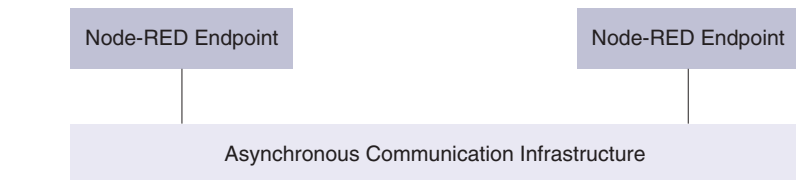


FIGURE 2. An endpoint-oriented architecture for IoT applications.

LISTING 1. STRUCTURES OF MESSAGES EXCHANGED BETWEEN THE ENDPOINTS IN THE LOGISTICS SCENARIO.

```

Merchant to Labeler: RequestLabel(orderID, address)
Merchant to Wrapper: RequestWrapping(orderID, itemID, item)
Wrapper to Packer: Wrapped(orderID, itemID, item, wrapping)
Labeler to Packer: Labeled(orderID, address, label)
Packer to Merchant: Packed(orderID, itemID, wrapping, label, status)
  
```

Unfortunately, Join aggregates the messages solely based on their sequence in each topic—and ignores their content—producing incorrect aggregations if the label and item do not actually correlate. For example, if Packer receives Labeled with orderID O_1 and Wrapped with orderID O_2 , Join would aggregate them and misdirect the wrapped item to the address from O_1 instead of O_2 .

Further, Join “consumes” each incoming message so it can appear in at most one output. Hence, only a single item can be associated correctly with a label. If any PO contains multiple items, Join matches the remaining items with labels from subsequent POs, causing all subsequent items to correlate with the wrong label.

Wait-paths based

Unlike Join, wait-paths (a community plugin node) supports a correlation field, here orderID. An explicit correlation field provides some support for correct matching.

However, a wait-paths node also “consumes” what it correlates and associates at most one item in a PO with the PO’s label. Thus, excess items are silently held (and eventually dropped) waiting for another matching label.

Custom

The Node-RED function node enables custom implementations. There are no limitations on the correctness or efficiency of a custom implementation, but such implementations would be low level and complex.

Shortcomings identified

First, although integrity constraints on correlation are explicit in the informal description of the scenario in the “Example IoT Scenario: Logistics” section, Listing 1 omits them. Representing constraints would enable reasoning about them computationally.

Second, traditional implementations entangle decision making internal to an endpoint (for example,

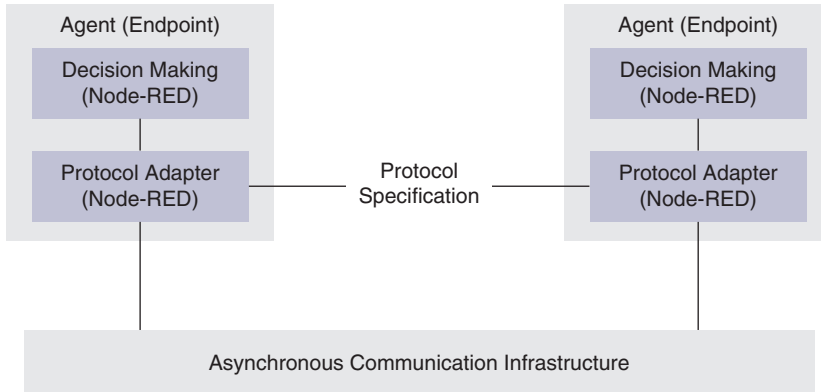


FIGURE 3. The PoT architecture for IoT applications.

LISTING 2. THE LOGISTICS PROTOCOL.

```

Logistics {
  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]
}
    
```

Packer deciding whether, when, and what item to pack) with public communication constraints (for example, to correlate items and labels).

Third, the endpoints may become inadvertently coupled by accounting for each other's implementation idiosyncrasies, for example, by accommodating one box per order but failing when the order is split.

THE PoT PROGRAMMING MODEL

The foregoing shortcomings motivate a new programming model.

Whereas traditional approaches focus on endpoints (as the “Challenges in Programming IoT Applications as Orchestrations” section indicates), PoT begins with a protocol: a specification that captures the interactions in an application via constraints on communications between the application's endpoints.

Figure 3 illustrates our protocol-based IoT application architecture. Here, an agent is an endpoint that adopts a role in a protocol and sends and receives messages to and from other agents in accordance with the protocol.

The Decision Making component implements an agent's private reasoning and relies on the Protocol Adapter to handle incoming and outgoing messages. The Adapter verifies whether messages respect the relevant causality and integrity constraints and discards any noncompliant messages—an outgoing message may not be emitted and an incoming message may not be made available to Decision Making. Notably, the Adapter is generic and need only be supplied with the protocol specification.

Specifying protocols

PoT adopts the Blindingly Simple Protocol Language (BSPL).¹⁰ Listing 2 specifies our logistics scenario in BSPL. Here, a protocol is a bag of message schemas. A protocol involves two or more roles; Logistics involves Merchant, Labeler, Wrapper, and Packer. A protocol has parameters; the idea is that when agents enact a protocol (by sending and receiving messages), they compute tuples of bindings for its parameters. Logistics' parameters are orderID, itemID, item, and status; enacting Logistics computes <orderID, itemID, item, status> tuples.

One or more protocol parameters are key parameters and jointly specify the key of the protocol; parameters orderID and itemID constitute the key in Logistics. A protocol's key specifies the integrity of the tuples computed by the protocol: at most, one tuple exists for a key binding. Integrity implies that a parameter may be bound only once relative to its key. Each distinct binding for a protocol's key corresponds to a distinct enactment of the protocol. A full tuple of bindings for the protocol's parameters corresponds to a complete enactment.

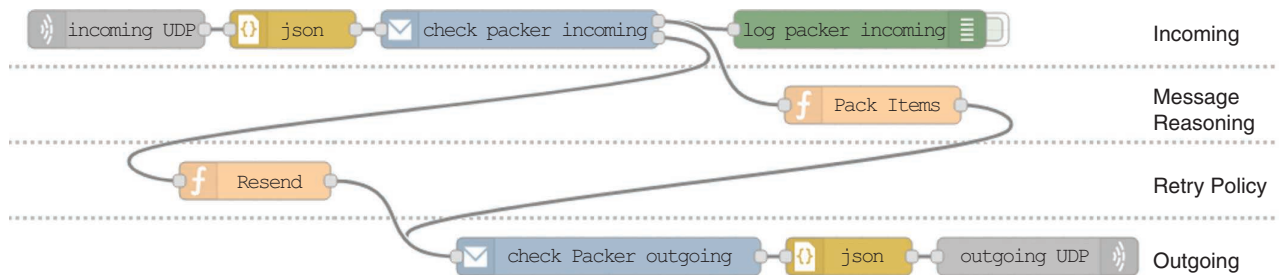


FIGURE 4. The PoT version of Packer flow depicting the architectural layers.

To capture causality constraints, a protocol's parameter is adorned `in` if the protocol depends on another protocol (via composition) for the parameter's binding; a protocol's parameter is adorned `out` if the protocol itself generates its binding. Logistics' parameters are all adorned `out`, meaning that enacting Logistics generates bindings for them all.

A message schema is an elementary protocol. Every message schema has a sender and a receiver role. Each message schema has parameters that are adorned either `in` or `out`, some of which constitute a key. The sender may send a message (instance) of the (message) schema if it knows (from prior interactions) the bindings of all parameters adorned `in` in the schema and does not know the bindings of any parameter adorned `out` in the schema. In sending the message, the sender can generate any binding for an `out` parameter. Thus, for example, Wrapper must know the bindings of `orderId`, `itemId`, and `item` before sending a Wrapped message but may generate a binding of wrapping.

In keeping with asynchrony, there are no constraints on when a message may be received; no ordering guarantees are required from the communication infrastructure. Parameter bindings become known to agents only through

message emissions and receptions; there is no shared storage.

Formalizing protocols yields three benefits. First, protocols separate decision making from interaction. Second, with BSPL specifically, integrity constraints capture correlation requirements declaratively. For example, the bindings of wrapping and label in a Packed message must be consistent with the bindings of `orderId` and `itemId` (Packed's key). Finally, as we discuss further in the next section, a clear specification of the communication constraints supports the implementation of agents by enabling the automatic generation of agent skeletons that enforce those constraints with the help of a generic protocol adapter.

IMPLEMENTING PoT AGENTS

Figure 4 depicts the implementation of a Packer agent according to the PoT model. The flow is separated into four layers: Incoming, Message Reasoning, Retry Policy, and Outgoing. The Incoming and Outgoing layers jointly constitute the protocol adapter component of the architecture but are separated because of the one-way flow programming style of Node-RED. Message Reasoning and Retry Policy together constitute the decision-making component; they handle new messages and duplicate messages, respectively.

Given a protocol, our tooling generates a skeleton consisting of the Incoming and Outgoing layers for every role in it. For every agent playing a role, the Message Reasoning and Retry Policy parts are added by the agent developer. We describe each layer below.

Incoming

The Incoming layer implements the reception half of an agent's protocol adapter. Nodes `incoming UDP` and `json` receive messages over UDP and decode them from JSON, respectively. Node `check Packer incoming` has access to the agent's history, which is a collection of messages that the agent has already observed. If an incoming message is not a duplicate and satisfies integrity, `check Packer incoming` adds it to the history and outputs it via the top output port. If the message is a duplicate, the node outputs it via the bottom output port without adding it to the history again. Integrity checking on reception is a defense against agents who may send messages in violation of integrity. Node `log Packer incoming` optionally logs received messages to the developer console.

Message Reasoning

The Message Reasoning layer captures an agent's reasoning, as the developer deems fit, as it reacts to received

messages and potentially generates outgoing messages. In Figure 4, the Packer implementation uses a single-function node `Pack Items` to send Packed messages in reaction to received `RequestLabel` and `RequestWrapping` messages. Outgoing messages may also be produced in response to other events; for example, Merchant could initiate Logistics based on POs in a database.

are not met within a specified time. Our Merchant implementation repeatedly resends the `RequestLabel` and `RequestWrapping` messages for an item every second until the corresponding Packed message is received.

However, resending a message only addresses the loss of that message; the loss of `Wrapped`, `Labeled`, or `Packed` would also prevent Merchant from receiving `Packed`. To further support recovery, the

support for agent retry policies reflects the end-to-end principle,⁷ providing application-specific assurance of correctness, without the overhead of redundant protections in lower layers of the infrastructure.

Outgoing

The Outgoing layer implements the emission half of an agent's protocol adapter. The layer checks every outgoing message, adding to the agent's history and sending over the network those that satisfy the relevant causality and integrity constraints, and dropping those that do not. In Figure 5, the `check Packer` outgoing node performs the checking. To support retry policies, the node allows sending duplicates; however, they are not added to the agent's history. Node `json` encodes the message in JSON and outgoing `UDP` sends it via UDP to the specified recipient.

EVALUATION

To demonstrate the advantages of PoT and its support for application-level retry policies, we evaluated four variations of our PoT-based implementation of the Logistics scenario:

- ▶ *Plain UDP*: PoT over UDP, without retries
- ▶ *UDP+Retry*: PoT over UDP, with the above-mentioned retry policies
- ▶ *MQTT0*: PoT over MQTT at quality-of-service (QoS) level 0
- ▶ *MQTT2*: PoT over MQTT at QoS level 2.

MQTT runs over TCP. QoS levels 0 and 2 mean that MQTT guarantees a message will be delivered at most once and exactly once, respectively.

We set up our experiment to compare the variations by their average

**THE PoT REPRESENTATION IS
UNIQUELY COMPATIBLE WITH
ASYNCHRONY AND NATURALLY
ADDRESSES CORRELATION PROBLEMS.**

Retry Policy

PoT also enables support for agent-specific retry policies for handling failure cases such as lost messages and reacting to duplicate incoming messages. If an agent expects to receive a future message in response to one that it sends, such as Merchant expecting a `Packed` message for each item, it can detect possible message loss when those expectations

other agents can resend relevant messages when they receive a duplicate. Our Node-RED implementation supports this pattern by providing a second output port on the reception checking node for handling duplicate messages. In our implementation of Packer, the `Resend` node handles duplicate `Labeled` or `Wrapped` messages and resends the corresponding `Packed` message if available. PoT's

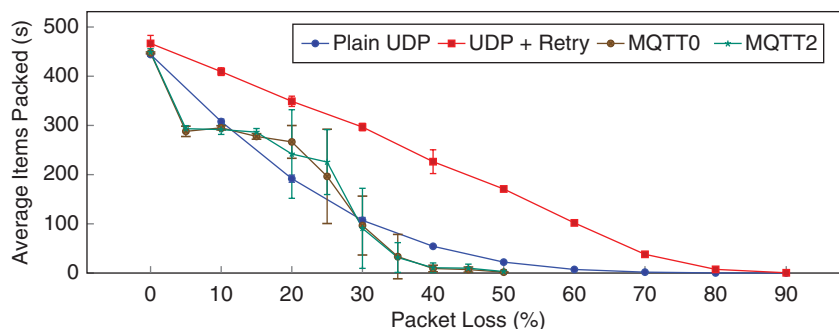


FIGURE 5. A comparison of different retry policies under different packet loss rates with respect to transaction completion.

throughput, defined as the rate at which enactments complete. We ran the experiment on a single Linux computer, using tc-netem to simulate random packet loss. For each variation and each level of packet loss, we ran 10 iterations of 1 min each. In each iteration, Merchant generates a new PO every 5 ms (simulating external customers), with the POs having a uniformly distributed number of items between one and four. Enactment timeouts were disabled, and the merchant's retry policy was configured to resend its messages every second until it received the corresponding Packed message.

Hypothesis. Under random packet loss, UDP+Retry completes more enactments over a given duration than both MQTT variations and Plain UDP.

Figure 5 provides the results of our experiment as a graph of the average throughput (enactments completed per second) over packet loss probability. Each plot represents the mean of 10 iterations, with error bars showing the sample standard deviation.

The results support our hypothesis. Plain UDP drops rapidly and consistently in throughput since any message loss prevents an enactment from completing. In contrast, UDP+Retry has a consistently higher throughput under packet loss. Since Plain UDP has less overhead and both initiate enactments at the same rate, the difference is in their reliability: UDP+Retry recovers some enactments that would otherwise be lost.

Both MQTT variations suffer an initial drop in throughput at low packet loss, possibly due to TCP backoff. Their throughput changes very little up to 20% loss, demonstrating reliability superior to Plain UDP and the benefits

of TCP's ability to batch multiple messages in one packet. However, both MQTT variations drop precipitously in throughput starting at 20% and fail to complete any enactments after 50% loss. At every packet loss rate, UDP+Retry has a higher throughput than either MQTT variation. Since TCP on Linux defaults to retrying packet transmissions for longer than the 1-min sample duration and UDP+Retry never gives up, the difference in throughput is not due to lost enactments but efficiency. UDP+Retry is more efficient than MQTT's infrastructure-level retry

solutions because it can detect losses without requiring message acknowledgments and no enactment blocks the progress of another.

PoT differs from existing work on protocol languages and deriving endpoint representations^{1,3} in its use of an information-based representation for protocols. The PoT representation is uniquely compatible with asynchrony and naturally addresses correlation problems. PoT is also of practical value as it enables the

ABOUT THE AUTHORS

SAMUEL H. CHRISTIE V is a Ph.D. student at North Carolina State University and a research associate at the School of Computing and Communications, Lancaster University. His research interests include decentralized and distributed systems and programming language design. Christie received his M.S. in computer science at North Carolina State University. Contact him at schrist@ncsu.edu.

DARIA SMIRNOVA was a research associate at the School of Computing and Communications, Lancaster University. Smirnova received her B.Sc. (with honors) in computer science from Lancaster University.

AMIT K. CHOPRA is a senior lecturer in the School of Computing and Communications, Lancaster University. His research interests include addressing challenges arising from autonomy and decentralization. Chopra received his Ph.D. from North Carolina State University. Contact him at amit.chopra@lancaster.ac.uk.

MUNINDAR P. SINGH is a professor of computer science and codirector of the Science of Security Lablet, North Carolina State University. His research interests include computational models of sociotechnical systems, including ways to reason about and program decentralized systems of autonomous agents. Singh received his Ph.D. in computer sciences from the University of Texas at Austin. He is a Fellow of IEEE and the Association for the Advancement of Artificial Intelligence. Contact him at singh@ncsu.edu.

generation of agent (endpoint) skeletons in Node-RED and, through them, the enforcement of protocol constraints.

PoT accommodates heterogeneity; an application may involve both PoT agents (implemented as described in the “Implementing PoT Agents” section) and non-PoT agents. PoT agents are guaranteed to be compliant with the protocol, but non-PoT agents can also be compliant. If a non-PoT agent is not compliant, however, the PoT agent’s adapter provides some protection by recognizing and rejecting incoming messages that fail integrity.

PoT’s basis in information protocols supports application-level retransmission of messages—a fault tolerance mechanism. PoT could be extended to support a wider and more sophisticated variety of such mechanisms. For example, retransmission could be adaptive: an agent could learn expected arrival times for messages from other agents relative to other messages and use that knowledge to determine when to resend a message. Future extensions should improve ease of use, flexibility, and robustness against various faults or malicious behavior until decentralized programming is no longer considered difficult but properly understood as the best way to improve fault tolerance and create scalable dynamic systems.

PoT opens up the possibility of expressing deeper expectations between parties, such as norms.² Just as PoT tackles expectations about the information exchanged, norms would capture the meaning of that information, such as whether one party committed to doing something for another or prohibited the other party from doing something. We could evaluate whether such a commitment or prohibition was satisfied or violated and

whom to hold to account for it. Such models can enable greater social intelligence, paving the path to superior programming models for autonomy, flexibility, and reusability. ■

ACKNOWLEDGMENTS

We thank the anonymous reviewers for helpful comments. Samuel H. Christie V, Daria Smirnova, and Amit K. Chopra were supported by the Engineering and Physical Sciences Research Council (grant EP/N027965/1). Samuel H. Christie V and Munindar P. Singh were partially supported by the National Science Foundation (grant IIS-1908374).

REFERENCES

1. M. Baldoni, C. Baroglio, and F. Capuzzimati, “A commitment-based infrastructure for programming socio-technical systems,” *ACM Trans. Internet Technol.*, vol. 14, no. 4, pp. 23:1–23:23, Dec. 2014. doi: 10.1145/2677206.
2. A. K. Chopra and M. P. Singh, “Custard: Computing norm states over information stores,” in *Proc. Int. Conf. Auton. Agents Multiagent Syst. IFAAMAS*, 2016, pp. 1096–1105.
3. A. Ferrando, M. Winikoff, S. Cranefield, F. Dignum, and V. Mascardi, “On enactability of agent interaction protocols: Towards a unified approach,” in *Proc. Eng. Multi-Agent Syst., 7th Int. Workshop*, 2019, pp. 43–63. doi: 10.1007/978-3-030-51417-4_3.
4. OASIS, “MQTT Version 3.1.1: OASIS Standard,” Burlington, MA, Oct. 2014. [Online]. Available: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf>
5. C. Peltz, “Web service orchestration and choreography,” *Computer*, vol.

36, no. 10, pp. 46–52, Oct. 2003. doi: 10.1109/MC.2003.1236471.

6. J. Postel, “User Datagram Protocol,” Internet Engineering Task Force, RFC 768, pp. 1–3, Aug. 28, 1980. [Online]. Available: <https://tools.ietf.org/html/rfc768>
7. J. H. Saltzer, D. P. Reed, and D. D. Clark, “End-to-end arguments in system design,” *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277–288, Nov. 1984. doi: 10.1145/357401.357402.
8. Z. Shelby, K. Hartke, and C. Bormann, “The Constrained Application Protocol (CoAP): Proposed standard,” Internet Engineering Task Force, RFC 7252, June 2014. [Online]. Available: <https://tools.ietf.org/html/rfc7252>
9. S. Sicari, A. Rizzardi, and A. Coen-Porisini, “Smart transport and logistics: A Node-RED implementation,” *Internet Technol. Lett.*, vol. 2, no. 2, p. e88, 2019. doi: 10.1002/itl2.88.
10. M. P. Singh, “Information-driven interaction-oriented programming: BSPL, the blindingly simple protocol language,” in *Proc. 10th Int. Conf. Auton. Agents Multiagent Syst.*, 2011, pp. 491–498.
11. M. P. Singh and A. K. Chopra, “The Internet of Things and multiagent systems: Decentralized intelligence in distributed computing,” in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, June 2017, pp. 1738–1747. doi: 10.1109/ICDCS.2017.304.



IEEE COMPUTER SOCIETY
DIGITAL LIBRARY

Access all your IEEE Computer Society subscriptions at

computer.org
/mysubscriptions