Pre-Proceedings of 8th International Workshop on Engineering Multi-Agent Systems
(EMAS 2020 @ AAMAS), 8-9 May 2020, Auckland, New Zealand.
Eds. Cristina Baroglio, Jomi F. Hübner, and Michael Winikoff.

Decentralized Programming for the Internet of Things

Samuel H. Christie V[1,2], Daria Smirnova[2], Amit K. Chopra[2], and Munindar P. Singh[1]

[1] North Carolina State University, Raleigh, NC 27695, USA
{schrist,singh}@ncsu.edu
[2] Lancaster University, Bailrigg, Lancaster, LA1 4YW, UK
{samuel.christie,d.smirnova,amit.chopra}@lancaster.ac.uk

Abstract. An Internet of Things (IoT) application is decentralized—i.e., it involves multiple asynchronously communicating loci of sensing, computing, and autonomy. Yet, current approaches do not model the decentralization explicitly, instead relying on endpoint specifications and ad hoc integration.

We address this mismatch via *Protocols over Things* (PoT), a decentralized approach for engineering IoT applications. PoT begins from an IoT application specified as an information protocol between (abstract) roles to be adopted by (concrete) agents. PoT compiles a protocol into specifications for each of its roles. To adopt a role, an agent implements the decision-making needed to enact the specification of that role.

We have implemented PoT on top of Node-RED, a popular IoT framework. We demonstrate how PoT simplifies implementation and avoids errors. Further, we empirically demonstrate that by supporting application-level retry policies, PoT provides improved performance over network-level delivery guarantees.

Keywords: Protocol · Autonomy · Interaction · Agents

1 Introduction

The value proposition of the Internet of Things (IoT) lies in enabling applications, e.g., smart health and smart cities, in which engagements between autonomous parties are supported by networked *things* that sense and communicate information about the environment. For example, in a health application, individuals would decide who to give access to health information from wearables but physicians would decide the diagnosis and request any further appointments based on the available information.

Such applications epitomize software architectures that are not only distributed (physically) but also *decentralized* since they comprise autonomous components, i.e., those who exhibit decision making. How can we engineer such applications?

Today's IoT programming approaches lack a computational representation of decentralization [15]. Popular frameworks such as Node-RED (https://nodered.org/) and Eclipse Kura (https://www.eclipse.org/kura/) model an IoT application as an orchestration that receives information from sensors, processes it by invoking a workflow, and effects actions in the environment. An orchestration is inherently central since it represents a single party's perspective [9]. Communication standards such as MQTT [6] and CoAP [12] support the distribution of resources in an IoT application but are agnostic to decentralization.

To enable capturing decentralization computationally, we contribute an approach called *Protocols over Things (PoT)*. Specifically, PoT captures a decentralized application via a *protocol* that specifies the communication constraints between the parties, abstracted as *roles* [15]. A role is analogous to an interface in object-oriented programming. For each role in the protocol, we compile out a specification that embodies the communication constraints applicable to the role. An agent *enacts* a role by supplying the decision logic for handling each received message and determining whether and when any message should be emitted. For concreteness and practicality, we adopt Node-RED as the target of our compilation.

We demonstrate PoT on a logistics scenario. We evaluate PoT by contrasting it with traditional Node-RED implementations. The similarity of the two approaches in all respects except protocols shows that the distinctions between them are crucial. First, we show that the PoT architecture and programming discipline (1) avoids subtle errors that common traditional implementations exhibit and (2) is simpler than a correct traditional implementation.

We demonstrate that PoT embodies the end-to-end principle [11], which argues against application endpoints relying on network-level delivery guarantees. Specifically, PoT supports deployment over asynchronous communication mechanims such as UDP (https://tools.ietf.org/html/rfc768). In fact PoT goes even further by addressing the challenge of message loss via application-level message retry policies. We experimentally demonstrate that a PoT implementation over UDP with application-level retry strategies compares favorably with the same implementation over MQTT.

2 Example IoT Scenario: Logistics

We adopt a warehouse logistics scenario [13], simplified to focus on relevant challenges.

Figure 1 illustrates this scenario. Here, Merchant, Wrapper, Labeler, and Packer are autonomous parties; the arrows indicate information flows. Merchant receives a purchase order (PO)—imagine an external customer. Each PO includes one or more items and a shipping address. Merchant sends the address to Labeler, who generates the appropriate shipment label to be attached to the shipping box. Labeler sends the generated label to Packer. Merchant sends information about the items to the Wrapper, who wraps the items appropriately for shipping (e.g., paper for durable items and bubble

wrap for fragile ones) and notifies PACKER they are ready. PACKER attaches 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 logic autonomously. For example, WRAPPER may select a wrapper based on current inventory and cost and LABELER may select a shipper based on reliability and cost. Conversely, autonomous agents may always choose not to send a message; e.g., WRAPPER may hold onto an item until it has the appropriate wrapping, or LABELER may reject invalid addresses. Further, these parties work concurrently and asynchronously based on information available to them.

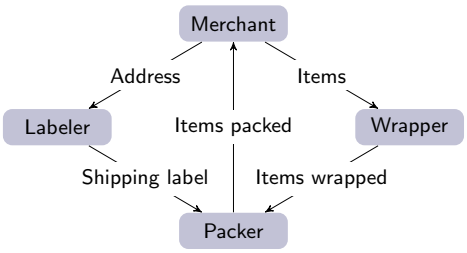

Fig. 1. Conceptual model of the logistics scenario, relative to a single purchase order (PO).

3 Challenges in Programming IoT Applications as Orchestrations

Node-RED is an interactive programming and execution environment for IoT applications. Node-RED runs in NodeJS (https://nodejs.org/). A Web interface presents a palette of function blocks called *nodes*. A user can construct a *flow*—technically an orchestration—by connecting those blocks via *wires* (from an output of a node to the input of another). Communication along the wires is asynchronous. Figure 2 illustrates a flow (its meaning is discussed below).

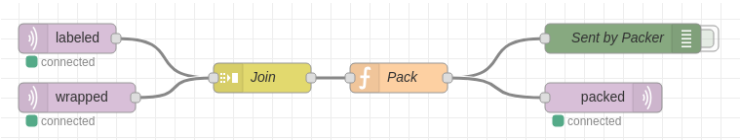

Fig. 2. Join-based Node-RED implementation for PACKER.

Each party's implementation is an *endpoint*. We use Node-RED to implement the scenario by encoding each endpoint as a flow that communicates with other

Listing 1. Schemas of messages exchanged between the parties' flows (the endpoints) in the logistics scenario.

```
//Sender to Receiver: MessageName(parameter 1,...,parameter n)

Merchant to Labeler: RequestLabel(orderID, address)

Merchant to Wrapper: Wrap(orderID, itemID, item)

Wrapper to Packer: Wrapped(orderID, itemID, wrapping)

Labeler to Packer: Labeled(orderID, address, label)

Packer to Merchant: Packed(orderID, itemID, wrapping, label, status)
```

endpoints via messaging over MQTT. Listing 1 gives details of the messages. Figure 3 illustrates the architecture schematically. Notably, there is no representation that captures the decentralization except as endpoints implemented in Node-RED.

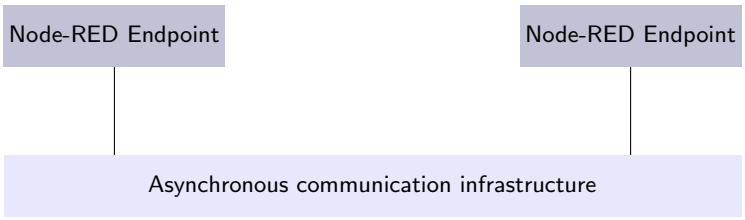

Fig. 3. Endpoint-oriented architecture for IoT applications.

We focus on Packer's flow, which is the most complex, correlating the labeled box with items (for a PO). Below, we describe three increasingly sophisticated Node-RED implementations of Packer and their advantages and limitations.

3.1 Join-Based

The `Join` node is a primary building block in Node-RED for aggregating information from multiple sources. Figure 2 illustrates a join-based implementation for Packer. The nodes `labeled` and `wrapped` represent MQTT subscriptions for *Labeled* and *Wrapped* messages. `Join` produces an aggregate message consisting of one message from each subscription. Node `Pack` transforms the aggregate to produce a *Packed* message, which is published via MQTT by the node `packed`.

A shortcoming of this join-based Packer implementation is that it aggregates labeled and wrapped messages as they come—whether or not they correlate. For example, if the first *Labeled* message Packer observes is for orderID 1 and the first *Wrapped* message it observes is for orderID 2, then `Join` aggregates

them, resulting in a *Packed* message with orderID 1, the label for orderID 1, but the wrapped item for orderID 2.

Further, `Join` "consumes" its incoming messages (that is, any incoming message can appear in at most one output). Hence, only a single item can be associated correctly with a label. Thus, if a PO contains multiple items, every item after the first item of this PO is most likely to be correlated with the wrong label.

3.2 Wait-Paths-Based

We implemented a more sophisticated version of PACKER that uses a `wait-paths` node for correlation. The essential difference from `Join` is that `wait-paths` supports specifying a correlation field, which in this implementation is set to orderID. Therefore, this implementation avoids incorrectly correlating labels.

However, this implementation suffers from the shortcoming of inadequate correlation: it associates at most one item with a label (`wait-path` too "consumes" what it correlates). That is, at most one item in a PO can be put into the box for that PO.

3.3 Custom Implementation

We implemented a correct Node-RED solution that packs all items in a PO in the labeled box for that PO. To this end, we applied the `function` node to write a custom correlation function in JavaScript. The code is available at https://gitlab.com/ieee-pot/ieee-pot.

3.4 Shortcomings Identified

We identify the following interaction-related shortcomings of the foregoing Node-RED implementations for decentralized applications.

First, communications between the parties must satisfy information integrity constraints, else incorrect correlation may arise. Notice that although the integrity constraints are explicit in the informal specification of the scenario, they are not represented formally in Listing 1. In fact, simpler integrity constraints—e.g., that for each orderID, MERCHANT and LABELER communicate a unique address and label, respectively—are not represented. Representing the constraints formally would enable reasoning about them computationally and inform programming models.

Second, lacking a representation of communication constraints, the foregoing implementations do not adequately support decision making in the flows. For example, a particular PACKER may exercise its autonomy by packing at most one item per box (thereby sending at most one *Packed* message). Another PACKER might not pack items for particular merchants. And yet another may pack all items. To support a variety of PACKER implementations, what would be desirable is that any implementation support correlating any item with its label and let

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 plug in its preferred decision-making policy about whether and how to pack an item.

Third, as the custom implementation shows, correct flows can be written by hand; however, it involves effort to get the logic right and each endpoint would have to implement it independently based on informally communicated requirements. And this is what leads to hidden coupling between the endpoints.

4 Specifying a Decentralized Application in PoT

Traditional approaches, as exemplified in Section 3, focus on the endpoints. In contrast, PoT begins with an interaction specification termed a *protocol*. A protocol captures the interactions in an application in terms of the constraints on communication between the endpoints.

PoT's specification language is *BSPL*, the Blindingly Simple Protocol Language [14]. Here, a protocol is a bag of messages, each with its sender and recipient roles, and a set of information parameters that it communicates. A role may send a message only if it satisfies dependencies based on the message's parameters that capture causal and integrity constraints. A role may receive a message whenever one arrives. Listing 2 specifies our scenario.

The *Logistics* protocol involves four roles. The parameters orderID, itemID, item, and status define this protocol's public interface. Here, orderID and itemID jointly form a key, meaning that for each combination of bindings of orderID and itemID, each of the other parameters may be bound to at most one value. Parameter bindings are observed only through the emission or reception of a message; all communication must be explicitly specified. Each tuple of bindings for the protocol's parameters represents a complete enactment.

Consider when a message may be sent. For each ⌜in⌝ parameter in that message, the sender must previously have observed that parameter—either by receiving or sending other messages. Thus WRAPPER must have prior knowledge of the item parameter before it can send *Wrapped*. Sending a message produces

a binding for each of its ⌜out⌝ parameters, so the message may be sent only if the sender has not yet observed that parameter. When MERCHANT sends *RequestLabel*, it produces a new binding of orderID.

Let's consider how PoT avoids the limitations of traditional implementations. First, the key constraints capture integrity requirements. Essentially, for any tuple of bindings for a message key (possibly composite, that is, consisting of multiple parameters), there can be at most one binding of its parameters. For example, for any binding of orderID, there may at most one binding label and address. Analogously, for any binding of (orderID, itemID), there can be at one binding of wrapping. This guarantees correct correlation in instances of *Packed*: it cannot be sent with bindings for wrapping and label different from what was observed previously.

Second, it separates the communication constraints from decision making. PACKER may choose to pack some subset of items for a PO, but it cannot send an incorrect *Packed* message. Third, a clear specification of the communication constraints enables generating a software specifications for the endpoint automatically, avoiding the need to write custom code. We discuss both these aspects in the following section.

5 PoT Platform

In the foregoing section, we described how to specify a decentralized application in terms of a protocol. In this section, we describe a platform for implementing decentralized applications that takes advantage of a protocol.

Figure 4 shows our protocol-based IoT application architecture. We use the term *agent* for a party's endpoint that adopts a role in a protocol. Each agent has a decision-making component that represents its policies. Agents interact on the basis of a protocol, captured by a specification and embodied in the Node-RED flows implementing each agents. We introduce two Node-RED nodes, `PoT-incoming` and `PoT-outgoing`, to support protocol-based interactions and provide clear separation between policy and protocol.

PoT-incoming The `PoT-incoming` node handles message reception, observing parameter bindings, and enforcing key constraints. It has two output ports, one for newly observed messages, and a second for duplicate messages that may indicate a need for retransmission and supports application-level retry policies.

A `PoT-incoming` node is configured with two pieces of information: a parameter list, and a timeout.
The parameter list specifies the enactment keys and message fields relevant to an agent. When a message is received, it is checked against any specified parameter constraints—specifically key constraints—and only recorded as *observed* if it is compatible with the agent's history. All valid, new messages are recorded and indexed by their enactment keys; duplicate messages are output via the second port. If a non-zero timeout is specified, the incoming node will delete enactments in which all messages are older than the timeout. The `PoT-incoming` node also

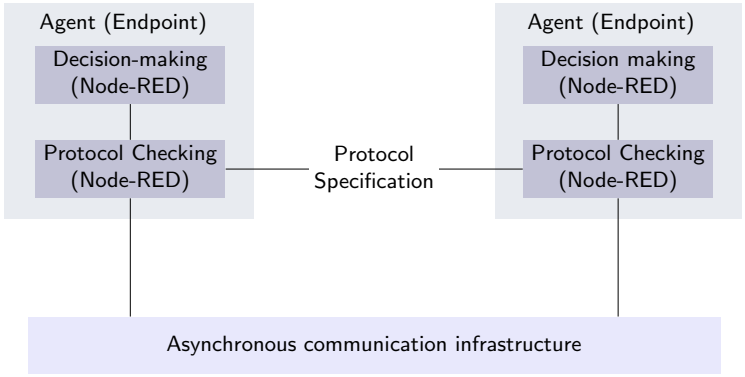

Fig. 4. PoT, that is, protocol-oriented architecture for IoT applications.

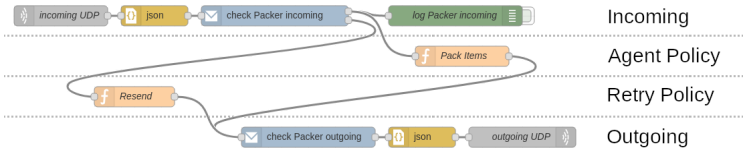

Fig. 5. PoT version of Packer flow, showing the architectural layers

attaches parameter bindings and references to previous messages from the same enactment.

PoT-outgoing `PoT-outgoing` checks messages before sending them, verifying that they match an enabled message schema. A `PoT-outgoing` node is configured with a list of message specifications, each with its own parameter list and a designated recipient. Each parameter may be adorned by one of ⌜out⌝, ⌜in⌝, or ⌜nil⌝, and one or more must be marked as key. The recipient information is used to dynamically route messages with one generic network output node instead of a separate path for each.

`PoT-outgoing` enforces all of the causality and integrity constraints, ensuring only correct messages are sent. Duplicates can be sent, but they are not observed a second time; this feature can be used to implement retry policies. Incorrect messages are dropped silently so `PoT-outgoing` can be used as a filter, simplifying policy code.

5.1 Implementing agents

Figure 5 shows the PoT implementation of Packer. The flow has been separated into four layers: Incoming, Agent Policy, Retry Policy, and Outgoing. The Incoming and Outgoing portions can be automatically generated as skeletons

from a protocol; the Agent Policy and Retry Policy are application specific and must be supplied by a developer.

5.2 Agent Decision Making

In PoT, agent decision-making policy is cleanly separated from interaction. The Packer takes each incoming message validated by `PoT-incoming`, and uses the attached enactment history instead of correlating the messages itself. If a label has been received and there are unpacked items, it packs them and outputs the notification, otherwise it waits for more information. The Packer did not need to implement any correctness checking or enactment correlation, it just processes the enactment data provided it by `PoT-incoming`.

The custom implementation mixed enactment management and correlation with its policy. To properly correlate items with boxes, it stored them all and checked the full index for matches. Although a custom, non-PoT implementation could certainly achieve equivalent correctness with greater performance, the lack of separation between policy and protocol tightens coupling and reduces reusability. Replacing the agent would require full knowledge of its implementation, since there would be no formal interaction specification to work with. Adapting or extending it to support additional scenarios could require rewriting the correlation logic, or reproducing its behavior as a pattern that could easily be forgotten or abused.

By separating policy from interaction, PoT supports adaptability, reusability, and autonomy. The agent's behavior can be changed with confidence that the adapters will ensure correctness, and adopting a different protocol is as simple as changing the adapter configuration.

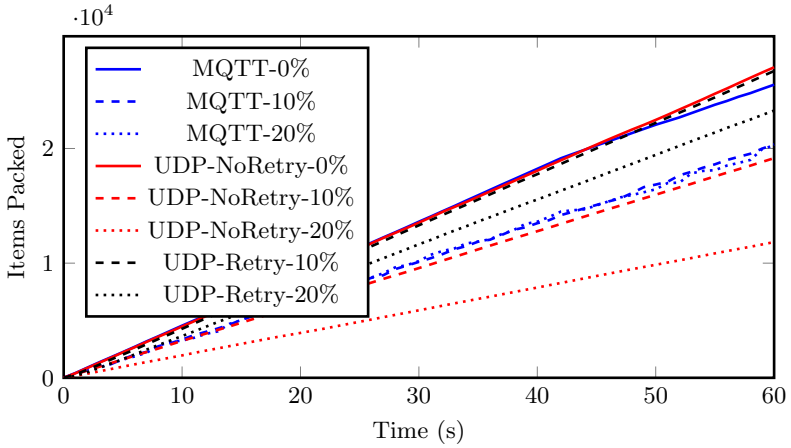

Fig. 6. A comparison of different retry policies under different packet loss rates (0, 10, and 20 percent loss).

5.3 Retry Policy

PoT also enables support for application-specific *retry policies*. Retry policies are a subset of agent policies specifically for handling failure cases such as lost messages. For example, the MERCHANT may fail to receive a *Packed* notification for an item if a relevant message was lost. In that case, MERCHANT has a retry policy that takes advantage of application-level understanding: the MERCHANT can tell that all of the messages have succeeded based on the enactment completion, without explicit acknowledgements; the other agents can safely resend messages in response to duplicates without worry of causing errors. More advanced policies could also account for agent failures.

In our PoT-based implementation, the MERCHANT records a queue of items for which it has initiated the *Logistics* protocol. If the enactment does not complete within a certain time, the MERCHANT restarts the enactment by sending duplicate copies of the *RequestLabel* and *RequestWrapping* messages. The `PoT-incoming` nodes output duplicate messages on a second port, so that retries can be handled separately. Each of the other agents reacts to a duplicate message by resending the message it produced in that enactment, without reprocessing them. Thus, if an enactment fails to complete because a message was lost between the agents, the messages are resent and propagated through the network until it completes.

PoT's support for agent retry policies is an application of the end-to-end principle [11], providing application-specific assurance of correctness, without the overhead of redundant protections in lower layers of the infrastructure.

6 Experimental Evaluation

To demonstrate the reliability PoT affords by enforcing integrity constraints and supporting application-level retry policies, we performed an experimental evaluation of three variations of our PoT-based implementation:

- PoT over UDP but no retries, meaning if a message is lost, then the corresponding enactment won't complete
- PoT over UDP with the above retry policies
- PoT over MQTT (relying on TCP for retries)

We ran the experiment on one physical machine, and tested the effects of emulated packet loss on the rate at which the three variations complete enactments.

Hypothesis 1 *Reliability. The reliability of a PoT implementation is the percentage of enactments that it completes. Because all of the messages in the protocol convey necessary information, any message loss will prevent an enactment from completing. Therefore, packet loss will reduce throughput.*

Our hypothesis is that the application-level retry policy will improve reliability, and complete more enactments over a given duration than the implementation over UDP without retries.

Hypothesis 2 *Performance. Even if two implementations have retry policies, it will still take longer to complete an enactment when a packet is lost than without retries because of the retransmission delay, reducing throughput. The effect on throughput depends on how many additional packets are required to recover from a loss.*

Our hypothesis is that the application-level retry policy will have better performance than the MQTT protocol under random packet loss, because MQTT over TCP requires a larger number of packets to complete an enactment.

6.1 Results

Figure 6 shows the results of our experiment, with the performance of each implementation represented as the cumulative number of items packed over the duration in seconds. Each plot shows the mean of five iterations at each of 0, 10, and 20 percent packet loss, sampled approximately every second since the first enactment was completed. The MERCHANT submitted purchase orders every 5 milliseconds, each with 1-4 items.

The results support both of our hypotheses. First, the UDP-Retry-10% performs almost as well as the other two at 0% loss, and 39% better than UDP-NoRetry-10%. The difference between UDP-Retry-20% and UDP-NoRetry-20% is even larger at 96%. This supports the reliability claim, clearly showing how damaging packet loss is without retries.

Second, UDP-Retry-10% performs 31% better than MQTT-10%, and UDP-Retry-20% performs 15% better than MQTT-20%. This shows that even the simple application-level retry policy we used can perform better than a network-level solution.

7 Conclusion

IoT applications are decentralized. We showed how PoT, the approach described in this paper, enables capturing the decentralization computationally via protocols. We demonstrated a PoT-based platform that supports implementing decentralized applications by ensuring the correctness of communications and providing a clean interface to plug in decision-making policies. We demonstrated the viability of enacting decentralized applications over UDP and in fact demonstrate performance advantages over MQTT. Notably, recent work [10] has emphasized the importance of developing efficient message-oriented middleware (MOM) for IoT.

There is a significant body of work on protocol languages and deriving endpoint representations from protocols, e.g., [1, 5]. PoT is novel in its use of an information-based representation for protocols, which is central to avoiding correlation problems. PoT is also directly of practical value as it generates endpoint representations in Node-RED.

There is increasing interest in IoT and agents. Ciortea et al. [3, 4] present an approach for designing scalable and flexible manufacturing systems, based

on multiagent system approach, automated planning and the notion of the Web of Things. They implement a prototypical production cell to demonstrate an approach that would achieve global collective intelligence for manufacturing. In contrast to PoT, which is based on protocols and can be used in settings of autonomous principals, their approach is based on planning, which may be thought of as a dynamic orchestration.

Padget, Vos, and Page [7] combine IoT with normative reasoning. Typically, such reasoning is either tightly coupled within agents or not present at all. Their approach makes normative frameworks a part of the architecture, loosely coupled with agents. The *deontic sensors* observe facts and generate institutional facts to be used by agents. Such normative reasoning can be created and deployed as a service by any web-client platform. PoT is complementary in its focus on operational protocols upon which normative meanings may be layered; that is, events generated by enacting protocols may use a substrate of facts for normative reasoning.

Due to IoT applications creating large volumes of data, the problem of handling, processing, and storing it has been addressed by Padgham and Winikoff [8]. They built a complex MAS architecture using the Prometheus methodology that does not depend on any language and can be applied in the IoT domain to provide reliable data storage solutions. In their work they describe low-level details of building a MAS architecture, including the creation of virtual machines, load balancing of the system, and collecting and authenticating data. Unlike Prometheus, PoT allows describing interactions between agents that represent high-level principals. However, the Prometheus methodology could potentially be applied together with PoT to help manage data created by agents.

Calegari et al. [2] bring the power of logic programming-based reasoning to the IoT domain. Their contribution is an architecture that consists of servers that connect with IoT devices, a Prolog-based reasoning engine that communicates with the servers to collect information, and end-user interfaces. Their architecture, even though it supports physical distribution, is conceptually unitary in that the engine encodes the application logic and interacts with the environment via sensors and user interfaces. Each agent in a protocol-based architecture could be implemented following their approach.

As the Internet of Things continues to grow in size, diversity, and importance, better approaches for designing and integrating decentralized systems will become even more necessary. An interesting direction would be to automatically generate retry and acknowledgement policies from protocol specifications, perhaps with the help of additional annotations.

Bibliography

[1] Baldoni, M., Baroglio, C., Capuzzimati, F.: A commitment-based infrastructure for programming socio-technical systems. ACM Transactions on Internet Technologies **14**(4), 23:1–23:23 (Dec 2014)

[2] Calegari, R., Denti, E., Mariani, S., Omicini, A.: Logic programming as a service (LPaaS): Intelligence for the IoT. In: 2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC). pp. 72–77. IEEE, Miami, US (May 2017). https://doi.org/10.1109/ICNSC.2017.8000070

[3] Ciortea, A., Boissier, O., Ricci, A.: Beyond physical mashups: Autonomous systems for the Web of Things. In: Proceedings of the 8th International Workshop on the Web of Things (WoT). pp. 16–20. ACM, Linz, Austria (Oct 2017)

[4] Ciortea, A., Mayer, S., Michahelles, F.: Repurposing manufacturing lines on the fly with multi-agent systems for the Web of Things. In: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 813–822. IFAAMAS, Stockholm (Jul 2018)

[5] Ferrando, A., Winikoff, M., Cranefield, S., Dignum, F., Mascardi, V.: On enactability of agent interaction protocols: Towards a unified approach. In: Proceedings of the 7th International Workshop on Engineering Multiagent Systems. p. to appear (2019)

[6] OASIS: MQTT 3.1.1 specification document (Oct 2014), OASIS Standard previously known as the Message Queuing and Telemetry Transport; http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf

[7] Padget, J., Vos, M.D., Page, C.A.: Deontic sensors. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18. pp. 475–481. International Joint Conferences on Artificial Intelligence Organization, Stockholm, Sweden (7 2018). https://doi.org/10.24963/ijcai.2018/66

[8] Padgham, L., Winikoff, M.: Prometheus: A practical agent-oriented methodology. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, chap. 5, pp. 107–135. Idea Group, Hershey, Pennsylvania (2005)

[9] Peltz, C.: Web service orchestration and choreography. IEEE Computer **36**(10), 46–52 (Oct 2003)

[10] Rausch, T., Dustdar, S., Ranjan, R.: Osmotic message-oriented middleware for the internet of things. IEEE Cloud Computing **5**(2), 17–25 (Mar 2018). https://doi.org/10.1109/MCC.2018.022171663

[11] Saltzer, J.H., Reed, D.P., Clark, D.D.: End-to-end arguments in system design. ACM Transactions on Computer Systems **2**(4), 277–288 (Nov 1984)

[12] Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). Tech. Rep. RFC 7252, Internet Engineering Task Force (IETF), Fremont, California (Jun 2014), proposed standard; https://tools.ietf.org/html/rfc7252

[13] Sicari, S., Rizzardi, A., Coen-Porisini, A.: Smart transport and logistics: A Node-RED implementation. Internet Technology Letters **2**(2), e88 (2019). https://doi.org/10.1002/itl2.88

[14] Singh, M.P.: 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). pp. 491–498. IFAAMAS, Taipei (May 2011). https://doi.org/10.5555/2031678.2031687

[15] Singh, M.P., Chopra, A.K.: The Internet of Things and multiagent systems: Decentralized intelligence in distributed computing. In: Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS). pp. 1738–1747. IEEE, Atlanta (Jun 2017). https://doi.org/10.1109/ICDCS.2017.304, Blue Sky Thinking Track