Protocol-Based Engineering of Microservices

Aditya K. Khadse[1], Samuel H. Christie V[2],
Munindar P. Singh[1], and Amit K. Chopra[3]

[1] North Carolina State University, USA
[2] Unaffiliated
[3] Lancaster University, Lancaster, UK
{akkhadse@ncsu.edu, shcv@sdf.org, mpsingh@ncsu.edu,
amit.chopra@lancaster.ac.uk}

Abstract. The *microservices* pattern is increasingly used in industry to realize applications in a decentralized manner, often with the help of novel programming models such as Microsoft-originated *Dapr*. Multiagent systems have typically been conceptualized as being decentralized. This naturally brings us to the question: Can multiagent software abstractions benefit the enterprise of realizing applications via microservices?

To answer this question, in this paper, we show how *interaction protocols*, a fundamental multiagent abstraction, can be applied toward realizing an application as a set of microservices. Specifically, we take a third-party application that exemplifies Dapr's programming model and reengineer it based on protocols. We evaluate the differences between our protocol-based implementation of the application and the Dapr-based implementation and find that our protocol-based implementation provides an improved developer experience in terms of cleaner, better-structured code. We conclude that (1) protocols represent a highly promising abstraction suited to the modeling and engineering of microservices-based applications and (2) Dapr augmented with a protocol-based programming model would be highly beneficial to the microservices enterprise.

Keywords: Decentralized systems · coordination · asynchronous messaging · multiagent systems · information protocols · programming models

1 Introduction

With the recent upsurge of cloud providers and affordable deployment solutions [20], large-scale software is increasingly written using microservices [29]. Microservices are motivated by loose coupling afforded by a decentralized application architecture. The microservices that constitute an application can be independently developed and maintained, possibly using heterogeneous technologies. Further, each microservice can be deployed in its own container and scaled independently in the cloud. By contrast, the components in a monolithic application [21] are tightly coupled.

A challenge with any decentralized architecture is coordination between its components. With products increasingly adopting the microservices architecture, programming models that facilitate microservices-based application development have emerged. Dapr [14] is a leading programming model, originally conceived within Microsoft, but now an open-source project. To support coordination, Dapr provides the abstractions of state stores, pub-sub brokers, and so on. Dapr is used across different industries by companies such as Alibaba Cloud [1] and Bosch [19]. Alibaba Cloud notes that adopting Dapr helped them integrate microservices written in different languages quickly. Bosch particularly mentions how it was easy to move to event-driven microservices while using Dapr.

The field of multiagent systems (MAS) has traditionally been concerned with decentralized architectures, and the connection between services and multiagent systems has been identified multiple times over the years [2, 22, 23, 26]. Particularly interesting are works on engineering MAS based on protocols [15–17].

Recent developments in engineering MAS have focused on the idea of information protocols [24, 33]. An information protocol models a decentralized MAS by specifying declarative information constraints on message occurrence. Information protocols are enacted by decentralized agents via Local State Transfer (LoST) [25]. Programming models based on information protocols includes Deserv [9], Bungie [8], Mandrake [10], and Kiko [11]. Kiko, in particular, represents a conceptual leap because it enables viewing and implementing an agent as a decision maker, its communications being its decisions.

In this paper, we model an existing Dapr application via information protocols and implement it using Kiko to highlight the benefits of a multiagent approach to microservices development. In particular, the benefits include better system modeling via protocols and attendant benefits such as verification; better structured and more correct code; fully decentralized implementations; and more loosely-coupled components.

2 Background

We now introduce information protocols, Kiko, and Dapr.

2.1 Information Protocols

Information protocols are declarative specifications of interaction between agents. A protocol specifies the *roles* (played by agents); a set of public parameters; optionally, a set of private parameters; and a set of messages. Each message specifies the sender, receiver, and its parameters. *Adornments* such as ⌜in⌝, ⌜out⌝, ⌜nil⌝ on parameters provide causal structure to the protocol. *Key* parameters identify enactments. Together, they constrain when messages may be sent. The adornment ⌜out⌝ for a parameter means in any enactment, the sender of the message can generate a binding (supply a value) for the parameter if it does already know it; ⌜in⌝ means that the parameter binding must already be known to the sender from some message in the enactment that it has already observed;

⌜nil⌝ means that the sender must neither already know nor generate a binding for the parameter. Each tuple of bindings for the public parameters corresponds to a complete enactment of the protocol. Thus, one can think of a protocol as notionally computing tuples of bindings via messaging between the roles.

Listing 1 is an example of an information protocol between a buyer, a seller, and a shipper for the purchase of an item.

Listing 1. The *Purchase* protocol [24].

```
Purchase {
  roles Buyer, Seller, Shipper
  parameters out ID key, out item, out price, out outcome
  private address, resp, shipped

  Buyer -> Seller: rfq[out ID, out item]
  Seller -> Buyer: quote[in ID, in item, out price]

  Buyer -> Seller: accept[in ID, in item, in price, out
      address, out resp]
  Buyer -> Seller: reject[in ID, in item, in price, out
      outcome, out resp]

  Seller -> Shipper: ship[in ID, in item, in address, out
      shipped]
  Shipper -> Buyer: deliver[in ID, in item, in address, out
      outcome]
}
```

Let's unpack how the protocol works. The name of the protocol is *Purchase* and Buyer and Seller are its roles. The parameters line specifies the tuple computed by a complete enactment of *Purchase*; parameterID is annotated *key*, meaning that it identifies tuples and the other parameters in the tuple are item, price, and outcome. These parameters are *public* and may be used toward composition with other protocols. A protocol may also have *private* parameters; here, address, resp, and shipped.

Every message has a sender, a receiver, a name, and a schema. The sequence in which these messages are written is unimportant. Causality is explicitly specified via parameter adornments. Specifically, to send a message instance of a particular schema, the bindings of its ⌜in⌝ parameters must already be known; the bindings of its ⌜out⌝ parameters must be generated in sending the instance and become known thereafter; and the bindings of its ⌜nil⌝ parameters must neither be known nor generated in the sending the instance. This means that in *Purchase*, Buyer can send an *rfq* at any point since all its parameter are adorned with ⌜out⌝. Once a *Seller* has received an *rfq*, it can send the corresponding *quote* since it knows the ID and item and it can generate price. And so on.

Messages can be made mutually exclusive (thus supporting choice within protocols) by adorning the same parameter as ⌜out⌝ in the messages. In the listing, the messages *accept* and *reject* both have resp adorned with ⌜out⌝. If

BUYER sends *accept*, it would have generated a binding for `resp`, which would mean that the sending of or *reject* would be disabled hence; and vice versa.

If *reject* is sent, the parameter `address` is never bound, and in effect, the messages *ship* and *deliver* will never be enabled. The enactment will be deemed as completed as all the needed parameters would be bound.

2.2 Kiko

Kiko is an information protocol-based programming model for agents. In other words, Kiko provides programming abstractions for implementing agents based on protocols.

Kiko takes to heart the idea that in a multiagent system, an agent's communications to others represent its decisions (it is in this sense that in multiagent systems, you have decentralized decision making). An agent is envisaged as running a loop in which upon the occurrence of certain events, it executes some business logic that may result in the making of new decisions, that is, the sending of messages to others.

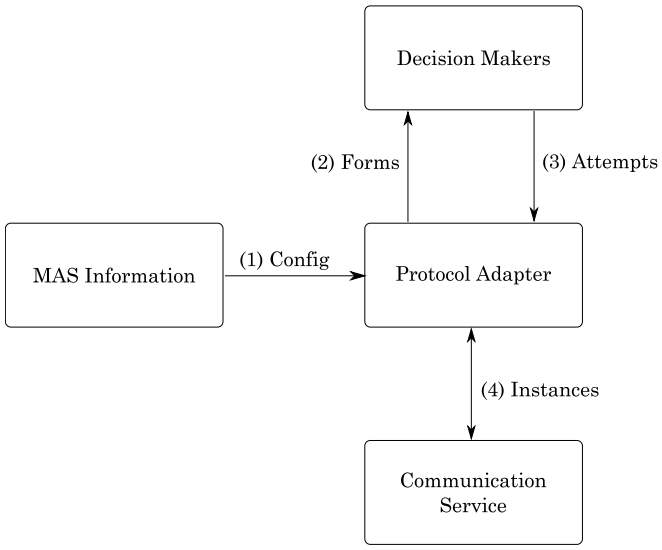

Fig. 1. The Kiko agent architecture [11].

To write an agent (Figure 1), an agent's programmer configures the agent with the multiagent systems it is playing roles in (based on protocols). In particular, it is configured with the identities of the other agents also playing roles in those multiagent systems and how to reach them over the network. Listing 2 shows an example of how configuration can be set up for MAS based on the protocol in Listing 1. We define one multiagent system named `SysName0` with

one agent for each role in the protocol. *Bob* is a BUYER, *Sally* is a SELLER and *Sheldon* is a SHIPPER.

Listing 2. A configuration of a multiagent systems using Kiko.

```
systems = {
    "SysName0": {
        "protocol": Purchase,
        "roles": {
            Buyer: "Bob",
            Seller: "Sally",
            Shipper: "Sheldon"
        }
    }
}
agents = {
    "Bob": [("192.168.0.1", 1111)],
    "Sally": [("192.168.0.2", 1111)],
    "Sheldon": [("192.168.0.3", 1111)]
}
```

Kiko's main abstraction is that of a *decision maker*. The programmer also writes a set of decision makers. A decision maker is a procedure written by the agent programmer that captures business logic. It is invoked upon the occurrence of a specified trigger event; it is supplied with the *enabled* (possible) decisions given the agent's communication history; and its body contains the logic to make some decisions (possibly none) from among the possible decisions. The possible decisions are known as *forms* and are supplied by the agent's protocol *adapter*, which keeps track of protocol enactments based on the messages the agent has observed. The name 'form' captures the idea that enabled decisions have their ⌜in⌝ parameters already filled in but the ⌜out⌝ parameters are yet to be bound, which is the job of the logic in the body. The fleshed-out forms are the message instances and are emitted on the wire by the adapter when the procedure returns. Message receptions are performed by the adapter transparently from the business logic. Kiko empowers programmers by enabling them to focus on business logic.

Listing 3 shows a decision maker for BUYER Bob. Its trigger is `InitEvent`, which represents the start of the agent. Thus, when the agent starts, this decision maker will be invoked by the adapter. The decision *rfq* is accessible as a form via the enabled argument. *Bob* is interested in a watch and so bind item in *rfq* to watch. The corresponding message instance is sent by the adapter to Sally (based on configuration) when the procedure returns.

Listing 3. Bob sending the *RFQ* message to Sally.

```
@adapter.decision(event=InitEvent)
def start(enabled):
    ID = str(uuid.uuid4())
    item = "watch"
    for m in enabled.messages(RFQ):
        m.bind(ID=ID, item=item)
```

Let's say that Sally has replied with a *quote* message providing the value of price. Now, Bob has to decide whether to *accept* or *reject* the quote. Listing 4 explains how an agent makes a decision.

Listing 4. Bob deciding whether to accept or reject a quote.

```
@adapter.decision
def decide(enabled):
    for m in enabled.messages(Buy):
        if m["price"] < 20:
            m.bind(address="1600 Pennsylvania Avenue NW",
                resp=True)
        else:
            reject = next(enabled.messages(Reject,
                ID=m["ID"]))
            reject.bind(outcome=True, resp=True)
```

The developer is in control of what is to be done at each junction of making a decision. Kiko provides this control through the use of sets of enabled messages. If an agent attempts to send both *accept* and *reject*, the messages would fail emission as the instances being sent are inconsistent with each other.

Because of our foundation in protocols (and roles), each agent may be implemented by a different programmer, thus highlighting Kiko's support for loose coupling. The steps below summarize the steps an agent programmer follows.

1. Define the configuration of the desired multiagent system (in Python).
2. Create an instance of an Adapter for each role using the class provided by Kiko (in Python).
3. Specify decision makers based on the information protocol using the previously written instance of an adapter (in Python). This specification of the decision makers ends up as an agent.
4. Start the agent (in Python).

The following details the services and API of the protocol adapter, a generic component of the programming model.

1. The protocol adapter is initiated within every agent, with the current agent's name, the configuration of the systems as well as the configuration of the other agents.
2. Depending on the protocol and the currently available information, certain decision makers are invoked by the protocol adapter. The protocol adapter provides forms, which are message instances with unbound parameters the decision maker can fill out.
3. These filled-out message instances are processed by the protocol adapter as attempts. The protocol adapter then checks the attempts for inconsistencies. In case of no inconsistencies, the message instances are successfully emitted; otherwise, they are dropped.
4. The protocol adapter relies on the communication service for transporting messages between agents. The default communication service is UDP, which is sufficient for enacting the information protocols. The adapter receives messages from other agents.

2.3 Dapr

Dapr is an event-driven runtime that promises resilient, stateless, and stateful microservices that interoperate. Dapr provides building blocks called *components*. Some popular types of components are:

- State Store: These components can be used as a database that is accessible to any Dapr application.
- PubSub Brokers: These components provide a system that supports the publishing of messages to a topic. Applications can then subscribe to these topics and receive published messages.
- Bindings & Triggers: These components enable Dapr applications to communicate to external services without integration of respective SDKs.

Dapr also provides a new type of component called *Pluggable components*. These components are not bundled as part of the Dapr runtime and run independently of it. The primary advantage of using a pluggable component is that it can be written in any language that supports gRPC.

3 Traffic Control Application

Traffic Control [32] is a sample application that emulates a traffic control system using Dapr. Figure 2 describes the application using a UML sequence diagram. It is inspired by the speeding-camera setup present on some Dutch highways. An entry camera is installed at the start of a highway and an exit camera is installed at a certain distance from the entry camera to capture vehicle license information. If a vehicle is going faster than the speed limit, the driver of the vehicle can be fined.

The time difference between an entry camera capturing a vehicle and an exit camera capturing the same vehicle will calculate the speed of the vehicle. Based on the speed of the vehicle, there is a decision to be made about whether the driver should be fined for driving over the speed limit.

3.1 Using Dapr

To develop this system in Dapr, four applications were created:

- Camera Simulation: A .NET Core console application that simulates passing cars.
- Traffic Control Service: A ASP.NET Core WebAPI application that defines two endpoints `/entrycam` and `/exitcam`
- Fine Collection Service: Another ASP.NET Core WebAPI application with only one endpoint `/collectfine` for collecting fines,
- Vehicle Registration Service: An ASP.NET Core WebAPI application with only one endpoint `/vehicleinfo/{license-number}`, which links a vehicle to its owner.

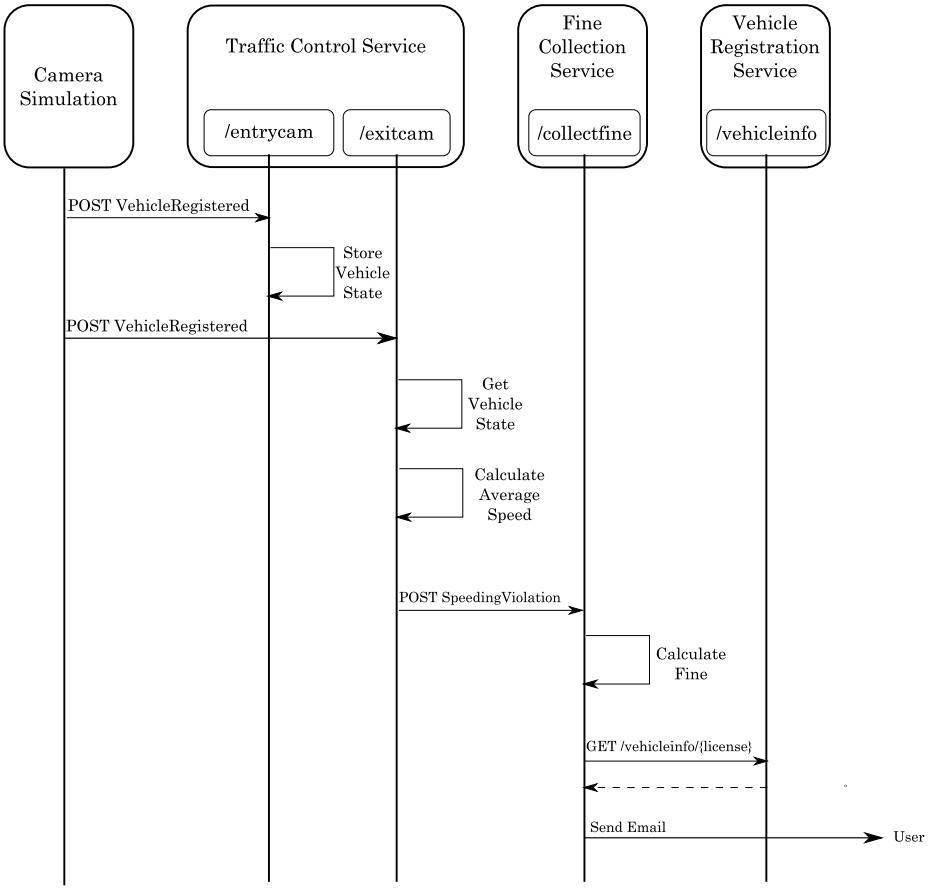

Fig. 2. A UML sequence diagram for the traffic control sample application.

A rundown of how this system operates follows:

1. Camera Simulation generates a random license number and sends a *VehicleRegistered* message (which contains the license number, the lane number, and the timestamp) to the `/entrycam` endpoint of Traffic Control Service.
2. The Traffic Control Service then stores the details in a database.
3. After a random interval of time, the Camera Simulation sends another *VehicleRegistered* message, but this time to the `/exitcam` endpoint of Traffic Control Service.
4. The Traffic Control Service then fetches the previously stored details and calculates the average speed of the vehicle.
5. If the average speed of the vehicle is greater than the speed limit, the Traffic Control Service sends the details of the incident to the endpoint `/collectfine` Fine Collection Service, where the fine is calculated.
6. The Fine Collection Service retrieves the email of the vehicle's owner by sending the details of the vehicle to the endpoint `/vehicleinfo/{license-number}` of the Vehicle Registration Service and sends the fine to the owner via email.

To enable the developer to focus on the business logic, Dapr provides components that are generic such as a database for storing the vehicle's information, providing an endpoint that can connect to an SMTP server that sends an email, and an asynchronous messaging queue that exchanges messages between the services.

Listing 5 shows how the `/exitcam` endpoint of the Traffic Control application deals with sending the fine. In particular, the endpoint is responsible for sending a `NotFound()` in case a vehicle that is not in the `_vehicleStateRepository` is detected by the exit camera.

Listing 5. The traffic control application's /exitcam endpoint.

```
1  [HttpPost("exitcam")]
2  public async Task < ActionResult >
       VehicleExitAsync(VehicleRegistered msg, [FromServices]
       DaprClient daprClient) {
3    try {
4      // get vehicle state
5      var state = await _vehicleStateRepository
6          .GetVehicleStateAsync(msg.LicenseNumber);
7      if (state ==
8        default (VehicleState)) {
9        return NotFound();
10     }
11
12     // update state
13     var exitState = state.Value with {
14       ExitTimestamp = msg.Timestamp
15     };
16     await _vehicleStateRepository
17         .SaveVehicleStateAsync(exitState);
```

```
18
19      // handle possible speeding violation
20      int violation = _speedingViolationCalculator
21          .DetermineSpeedingViolationInKmh(
22              exitState.EntryTimestamp,
23              exitState.ExitTimestamp.Value
24          );
25
26      if (violation > 0) {
27        var speedingViolation = new SpeedingViolation {
28          VehicleId = msg.LicenseNumber,
29            RoadId = _roadId,
30            ViolationInKmh = violation,
31            Timestamp = msg.Timestamp
32        };
33
34        // publish speedingviolation (Dapr pubsub)
35        await daprClient.PublishEventAsync("pubsub",
              "speedingviolations", speedingViolation);
36      }
37
38      return Ok();
39    } catch (Exception ex) {
40      return StatusCode(500);
41    }
42  }
```

3.2 Using Kiko

To implement the Traffic Control system in Kiko, we initially need to create a protocol that can accommodate all of our requirements. Listing 6 shows an example of a protocol that would enable us to fulfill the requirements and is supported by the tooling.

Listing 6. The *TrafficControl* protocol.

```
TrafficControl {
    roles EntryCam, ExitCam, FineCollector, VehicleMngr
    parameters out regID key, out entryTS, out exitTS, out
        email
    private amount, avgSpeed, query

    EntryCam -> ExitCam: Entered[out regID, out entryTS]
    ExitCam -> FineCollector: Fine[in regID, in entryTS, out
        exitTS, out avgSpeed]
    FineCollector -> VehicleMngr: Query[in regID, in
        entryTS, in avgSpeed, out query]
    VehicleMngr -> FineCollector: Result[in regID, in
        entryTS, out email]
}
```

Let's unpack how this protocol works. The roles involved would be EntryCam, ExitCam, FineCollector, VehicleMngr. The parameters necessary for the completion of an enactment are regID which stands for registration ID, entryTS which stands for entry timestamp, exitTS which stands for exit timestamp, and outcome. Private parameters that may or may not be bound are amount, avgSpeed which stands for average speed, and query.

The first message that will be sent out is *Entered*. This denotes the EntryCam alerting the ExitCam that a vehicle has entered the highway. The next message that will be sent out is *Fine*. This is where the decision maker defined by the developer will come into play. Listing 7 shows one such implementation of the decision maker. The code is written in Python by the developer and uses the Kiko library [7]. Constants in uppercase are part of the configuration. Currently, the entry camera is simulated by a trigger event that is invoked at random times. The exit cam is simulated by adding a random amount of time to a known entry timestamp. This could easily be replaced with a blocking call to the method that would wait to observe a vehicle and continue in case the vehicle matches the registration. An observation that could be made is that it is unnecessary to explicitly store the exit timestamp as every observation is stored in the local store.

Listing 7. A decision maker for the exit camera.

```
@adapter.decision(event=VehicleExit)
async def check_vehicle_speed(enabled, event):
    for m in enabled:
        if m.schema is Fine and m["regID"] == event.regID:
            avgSpeed = DISTANCE / (event.ts - m["entryTS"])
            if avgSpeed > SPEED_LIMIT:
                m.bind(exitTS=event.ts, avgSpeed=avgSpeed)
                return m
```

We create a single decision maker for deciding whether *Fine* message should be sent next. If the Fine Collector receives the message *Fine*, it then retrieves the details of the owner of the vehicle from the Vehicle Manager and sends the email detailing the fine. The code for this implementation can be found on `https://gitlab.com/masr/kiko-traffic-control`. Figure 3 shows the UML sequence diagram for our implementation using Kiko.

Internal computations are omitted from the UML diagram. For example, the average speed is calculated by Exit Cam, hence a message like Exit is not explicit in the protocol.

4 Evaluation

We evaluate the implementation based on the differences in the Kiko and Dapr implementations of the scenario.

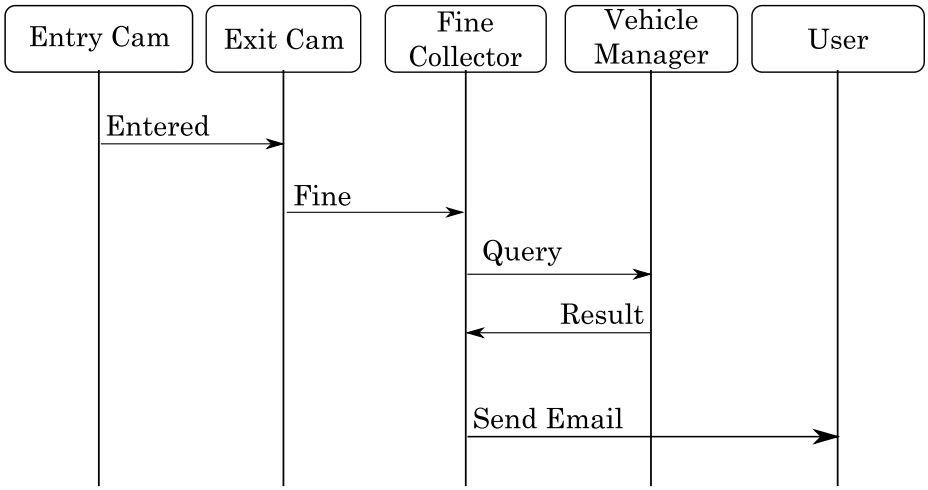

Fig. 3. A UML sequence diagram for our traffic control sample application written using Kiko.

4.1 Protocol Specifications

Protocols are at the heart of both implementations. The Kiko implementation relies on the formal specification of the protocol. The protocol can be verified statically for properties such as safety and liveness. Further, an adapter takes the protocol as input (serving as a runtime) and enables implementing the agent based on the protocol. In the Dapr-based implementation, the protocol is specified only informally using UML interaction diagrams. They afford neither verification nor a protocol-based programming model.

4.2 Typing and Structuring of Agent Implementations

In Kiko, the information protocol already captures crucial domain aspects related to the interaction, such as the entry and exit identifiers, registration identifiers, and so on. These domain-related aspects are not modeled or are captured only in low-level data structures in the Dapr implementation (line 6 in Listing 5). Kiko can enforce integrity checking based on identifiers that are annotated as key. In Dapr, the agent developer has to write such integrity-checking code.

Kiko shines in structuring the agent implementation and focusing the developer on writing the business logic, with fewer possibilities for errors in decision making. Its notion of forms is particularly helpful as it provides decisions (messages) that have known information already filled in and points the developers to writing code that generates the missing information. By contrast, Dapr developers must construct entire messages by hand (line 27 in Listing 5), which introduces possibilities for errors.

The Dapr implementation contains code for getting and updating the state (Lines 5–17) that doesn't appear in the Kiko implementation because the adapter

automatically maintains the state. Further, the Dapr implementation contains code for the 'error' of exit being recorded but there is no record of entry (Lines 7–9). Such error handling doesn't appear in the Kiko decision maker in Listing 7: if the exit camera doesn't hasn't received the message denoting entry, then the corresponding *Fine* form will not appear in the set of forms supplied by the adapter. Whether exits correlate with entries is something worth keeping track of as missing entries or exits may indicate problems with the cameras; however, such code need not have to be in the exit camera.

Dapr's implementation relies on the developers being responsible for integrating the endpoints. It is possible that an external agent tries to send an invalid request to an endpoint. Kiko's implementation on the other hand only relies on **agents conforming to the protocol**. Even if an external agent attempts to push a message to the agent, if the history does not match with the message, it will be ignored by Kiko.

4.3 Decentralization and Loose Coupling

Although microservices aspire to decentralization, the Dapr implementation actually relies on a shared state between the entry and exit camera endpoints. Specifically, the entry camera endpoint stores information about the entry in a shared store which is then retrieved by the exit camera endpoint to update it with the exit information and to calculate the average speed (Lines 3–15). By contrast, there is no shared state between Kiko agents. The only way for Kiko agents to share information is to transfer their local state via messaging.

Being able to independently implement and maintain endpoints is evidence of loose coupling. Since the Dapr implementation is not based on a high-level system model (the information protocol) and to interoperate the endpoint developers would have to share code, loose coupling is better supported by Kiko than by Dapr.

For asynchronous communication between microservices, the Dapr implementation relies on publish-subscribe communication via message queues. The Kiko implementation by contrast uses UDP (a lossy, unordered communication service) as the underlying communication service highlighting the fact that ordered message delivery is unnecessary. Other information protocol-based works [10] have shown how agents can deal with message loss.

5 Discussion

Based on this evaluation, we can conclude that the Kiko implementation provides a better developer experience and is better suited to decentralization and loose coupling.

Since the Kiko implementation of the traffic control application does not rely on message ordering for processing, it is possible that the exit camera agent records `VehicleExit` occurs prior to the reception of the *Entered* message. In this case, as the *Fine* message would not be enabled and therefore there is no

possibility of a fine being issued. This may not be the desired effect but can be remedied by writing a decision maker that iterates over enabled messages on the reception of the *Entered* message.

Using the microservice architecture also requires a fair amount of knowledge dealing with deploying different services. A dedicated DevOps team was found to be necessary for software that followed microservice architecture [28]. With only about 10% of respondents claiming to be a DevOps specialist in the 2022 Developer Survey [27] by Stack Overflow, developers end up being the ones deploying the applications. Using Dapr, this job becomes easier to deal with when using applications written in different languages.

We posit that the conceptual integration between MAS and web architecture would facilitate the construction of multiagent systems that are widely distributed and inherit architectural properties such as scalability and evolvability [13]. The integration of Kiko with Dapr would enable the users to build a MAS that has the benefits of microservice architecture such as scalability but also the benefits of observability and secret management. Further, this conceptual integration can also lead to the resulting system being close to a Hypermedia MAS [12].

Multiagent systems need to provide an account of what happened during an abnormal situation [4, 6]. Kiko would provide the protocol as a blueprint while Dapr would provide robust tooling for the observation of intercommunication of the microservices.

Microservices of the future should look at a move towards asynchronous communication [18] and this idea is supported in information protocols through their causal nature and their ability to operate on lossy, unordered protocols such as UDP. A not surprising lesson learned in an exploratory study of promises and challenges in microservices [31] was that changes that break the API should be discouraged. The use of information protocols enables the developers to version interactions between the microservices. Since the protocol file defines all valid interactions between microservices, it also defines implemented interactions between microservices of a release ready for the production environment. Timeouts within a microservice system is a problem that is remedied by using a circuit breaker [30] but has the tradeoff of requiring an update on all microservices. With the use of information protocols, we move away from synchronous communication and remove the need for timeouts and consequently circuit breakers.

5.1 Future Work

To fully obtain the benefits of Kiko under Dapr, the ideal solution would be to build a Pub/Sub–based pluggable component in Python, that uses Kiko to work on the messages. The queues that would be created as part of the Pub/Sub communication between applications, must have their messages assessed through Kiko. This way we would emulate Kiko's protocol adapter within this pluggable component and send forms to be filled out as attempts by the applications. Since Kiko can act as a verification agent, it would enable runtime verification of the developed MAS similar to existing solutions for other MASs [5]. Verification at

design time [3] would also be possible as endpoints within Dapr applications are registered and known to Dapr prior to any communication between applications.

Currently, Kiko can invoke methods on the reception of a particular message or event or the enablement of a particular message. We cannot invoke a method only if multiple events are received or multiple messages are received. A future iteration could support how decision makers may be invoked when a specified set of events (i.e., messages) are received.

In cases where enactments are not fulfilled, the messages stay in the local store forever. These dangling enactments would eventually prevent storing new enactments. An automated job that gets rid of these enactments can be added to be run after a fixed time interval. As all enactments are linked via the key parameters, it is easy to identify what messages must be discarded.

Acknowledgments

We thank the EMAS 2023 reviewers and audience for their helpful comments. We acknowledge support from the UK EPSRC (grant EP/N027965/1) and the US NSF (grant IIS-1908374).

References

1. Ao, S.: How Alibaba is using Dapr, https://blog.dapr.io/posts/2021/03/19/how-alibaba-is-using-dapr/, accessed 19 February 2023
2. Baldoni, M., Baroglio, C., Chopra, A.K., Desai, N., Patti, V., Singh, M.P.: Choice, interoperability, and conformance in interaction protocols and service choreographies. In: Proceedings of the 8th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 843–850. IFAAMAS, Budapest (May 2009). https://doi.org/10.5555/1558109.1558129
3. Baldoni, M., Baroglio, C., Martelli, A., Patti, V.: A priori conformance verification for guaranteeing interoperability in open environments. In: Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC). Lecture Notes in Computer Science, vol. 4294, pp. 339–351. Springer, Chicago (Dec 2006). https://doi.org/10.1007/11948148_28
4. Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S.: Accountability in multiagent organizations: From conceptual design to agent programming. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **37**(1), 7 (Jun 2023). https://doi.org/10.1007/s10458-022-09590-6
5. Briola, D., Mascardi, V., Ancona, D.: Distributed runtime verification of jade multiagent systems. In: Camacho, D., Braubach, L., Venticinque, S., Badica, C. (eds.) Intelligent Distributed Computing VIII. pp. 81–91. Springer International Publishing, Cham (2015)
6. Chopra, A.K., Singh, M.P.: Accountability as a foundation for requirements in sociotechnical systems. IEEE Internet Computing (IC) **25**(6), 33–41 (Sep 2021). https://doi.org/10.1109/MIC.2021.3106835
7. Christie, S.: Kiko, https://gitlab.com/masr/bspl/-/tree/kiko/, accessed 15 February 2023

8. Christie V, S.H., Chopra, A.K., Singh, M.P.: Bungie: Improving fault tolerance via extensible application-level protocols. IEEE Computer **54**(5), 44–53 (May 2021). https://doi.org/10.1109/MC.2021.3052147
9. Christie V, S.H., Chopra, A.K., Singh, M.P.: Deserv: Decentralized serverless computing. In: Proceedings of the 19th IEEE International Conference on Web Services (ICWS). pp. 51–60. IEEE Computer Society, Virtual (Sep 2021). https://doi.org/10.1109/ICWS53863.2021.00020
10. Christie V, S.H., Chopra, A.K., Singh, M.P.: Mandrake: Multiagent systems as a basis for programming fault-tolerant decentralized applications. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **36**(1), 16:1–16:30 (Apr 2022). https://doi.org/10.1007/s10458-021-09540-8
11. Christie V, S.H., Singh, M.P., Chopra, A.K.: Kiko: Programming agents to enact interaction protocols. In: Proceedings of the 22nd International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 1–10. IFAAMAS, London (May 2023)
12. Ciortea, A., Boissier, O., Ricci, A.: Engineering world-wide multi-agent systems with hypermedia. In: Weyns, D., Mascardi, V., Ricci, A. (eds.) Engineering Multi-Agent Systems. pp. 285–301. Springer International Publishing, Cham (2019)
13. Ciortea, A., Mayer, S., Gandon, F., Boissier, O., Ricci, A., Zimmermann, A.: A decade in hindsight: The missing bridge between multi-agent systems and the world wide web. In: Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems. p. 1659–1663. AAMAS '19, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2019)
14. Dapr: Dapr – Distributed Application Runtime (2019), https://dapr.io/, accessed 14 February 2023
15. Desai, N., Mallya, A.U., Chopra, A.K., Singh, M.P.: Interaction protocols as design abstractions for business processes. IEEE Transactions on Software Engineering **31**(12), 1015–1027 (Dec 2005). https://doi.org/10.1109/TSE.2005.140
16. Desai, N., Mallya, A.U., Chopra, A.K., Singh, M.P.: OWL-P: A methodology for business process development. In: Agent-Oriented Information Systems III, 7th International Bi-Conference Workshop, AOIS2005, Utrecht, Netherlands, July 26, 2005, and Klagenfurt, Austria, October 27, 2005, Revised Selected Papers. pp. 79–94. No. 3529 in Lecture Notes in Computer Science, Springer, Berlin (2006). https://doi.org/10.1007/11916291_6
17. 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 Multi-Agent Systems (EMAS). Lecture Notes in Computer Science, vol. 12058, pp. 43–64. Springer, Montréal (May 2019). https://doi.org/10.1007/978-3-030-51417-4_3
18. Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S.: Microservices: The journey so far and challenges ahead. IEEE Software **35**(3), 24–35 (2018). https://doi.org/10.1109/MS.2018.2141039
19. Microsoft: Bosch builds smart homes using Dapr and Azure, https://customers.microsoft.com/en-us/story/1435725395247777374-bosch-builds-smart-homes-using-dapr-azure, accessed 19 February 2023
20. PwC: Cloud business survey, https://www.pwc.com/us/en/tech-effect/cloud/cloud-business-survey.html, accessed 14 February 2023
21. Richardson, C.: Monolithic architecture pattern, https://microservices.io/patterns/monolithic.html, accessed 8 February 2023

22. Singh, M.P.: Synthesizing distributed constrained events from transactional workflow specifications. In: Proceedings of the 12th International Conference on Data Engineering (ICDE). pp. 616–623. IEEE, New Orleans (Feb 1996). https://doi.org/10.1109/ICDE.1996.492212
23. Singh, M.P.: Distributed enactment of multiagent workflows: Temporal logic for web service composition. In: Proceedings of the 2nd International Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 907–914. ACM Press, Melbourne (Jul 2003). https://doi.org/10.1145/860575.860721
24. 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
25. Singh, M.P.: 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). pp. 57–64. IEEE Computer Society, Washington, DC (Jul 2011). https://doi.org/10.1109/ICWS.2011.48
26. Singh, M.P., Chopra, A.K., Desai, N.: Commitment-based service-oriented architecture. IEEE Computer **42**(11), 72–79 (Nov 2009). https://doi.org/10.1109/MC.2009.347
27. Stack Overflow: Stack Overflow 2022 Developer Survey, https://survey.stackoverflow.co/2022/, accessed 14 February 2023
28. Taibi, D., Lenarduzzi, V., Pahl, C.: Continuous architecting with microservices and DevOps: A systematic mapping study. In: Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER): Revised Selected Papers. Communications in Computer and Information Science, vol. 1073, pp. 126–151. Springer, Funchal, Madeira, Portugal (Mar 2018). https://doi.org/10.1007/978-3-030-29193-8_7
29. Thönes, J.: Microservices. IEEE Software **32**(1), 116–116 (2015). https://doi.org/10.1109/MS.2015.11
30. Tighilt, R., Abdellatif, M., Moha, N., Mili, H., Boussaidi, G.E., Privat, J., Guéhéneuc, Y.G.: On the study of microservices antipatterns: A catalog proposal. In: Proceedings of the European Conference on Pattern Languages of Programs 2020. EuroPLoP '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3424771.3424812
31. Wang, Y., Kadiyala, H., Rubin, J.: Promises and challenges of microservices: an exploratory study. Empirical Software Engineering **26**(4), 63 (May 2021). https://doi.org/10.1007/s10664-020-09910-y
32. van Wijk, E., Molenkamp, S., Hompus, M., Kordowski, A.: Dapr traffic control sample, https://github.com/EdwinVW/dapr-traffic-control, accessed 15 February 2023
33. Winikoff, M., Yadav, N., Padgham, L.: A new Hierarchical Agent Protocol Notation. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **32**(1), 59–133 (Jan 2018). https://doi.org/10.1007/s10458-017-9373-9