Protocol-Based Engineering of Microservices

Aditya K. Khadse[1], Samuel H. Christie[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. In industry, the microservice pattern is increasingly used to realize decentralized applications, often with the help of programming models such Dapr. Multiagent systems have always been thought of as being decentralized. Can multiagent software abstractions benefit the engineering of microservices?
In this paper, we show how that interaction protocols, a fundamental multiagent abstraction, facilitates the engineering of applications in a manner that respects the microservices pattern. To make our case, we take a third-party application that exemplifies Dapr's programming model and reengineer it in based on protocols. We evaluate the differences between our protocol-based implementation and the Dapr implementation and find that our protocol-based implementation provides an improved developer experience along with a verifiable causal structure to the business process. We conclude that augmenting Dapr with protocol-based programming abstractions would be high beneficial to the microservices enterprise.

Keywords: microservices · Dapr · multiagent systems · information protocols · agent programming

1 Introduction

With the recent upsurge of cloud providers and affordable deployment solutions [16], most large-scale software is written using microservices [22]. Microservices are motivated from loose coupling afforded by a decentralized application architecture: The microservices that constitute an application can be independently maintained, deployed, and scaled (in the cloud). By contrast, the components in a monolithic application [17] 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 [13] is one such leading programming model, originally conceived within Microsoft, but now an open source project. To support coordination, Dapr provides the abstractions of state stores, pubsub brokers, and so on.

Dapr is used across different industries by companies such as Alibaba Cloud [1] and Bosch [15]. 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. Recent developments have focused on the idea of information protocols. An information protocol [18] models a decentralized MAS by specifying declarative information constraints on message occurrence. Information protocols are enacted by decentralized agents via Local State Transfer (LoST) [19]. Programming models based on information protocols includes Deserv [8], Bungie [7], Mandrake [9], and Kiko [10]. Kiko in particular enables developers to focus on writing the decision makers of agents.

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 are:

- All interactions between microservices can be asynchronous.
- Safer interactions as they will always have valid information.
- Assign ownership of parameters to corresponding agents.
- Provide documentation of roles of microservices, their interactions, ownership of parameters, and flow of control.

We then evaluate the implementation by comparing it to the existing application, report our findings, and discuss some ideas for future work.

2 Background

In this section, we describe 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 know already 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.

```
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 first line mentions the name of the protocol, which is *Purchase* in this case. Next, we list out the roles that will be involved in the protocol, BUYER and SELLER. We then list out the parameters that are part of the messages being sent between the roles. Here, the key parameter is ID and other parameters are item, price, and outcome. The private parameters are address, resp, and shipped. Beyond this, every line defines a message.

Every message has a sender, a receiver, a name, and a schema. The sequence in which these messages are written is unimportant. The causal inference is made using the adornments of parameters. If a message has all of its parameters adorned with ⌜in⌝ bound, then that message becomes *enabled* meaning it can be sent once the agent fills all the parameters with ⌜out⌝ adornment. Hence, the first message that can be sent in this protocol becomes *rfq*, as all its parameter are adorned with ⌜out⌝ which can be created by the sender. The next message that can be sent is *quote*, as its parameters ID and item were bound by the *rfq* message that was received, and the parameter price.

Choice within protocols can be supported by providing messages that conflict with each other. In the listing, we can see that the messages *accept* and *reject* both have resp as a parameter adorned with ⌜out⌝. An agent cannot rebind a parameter. Hence, the agent has to make a choice where either *accept* or *reject* is sent. If *reject* is sent, the parameter address is never bound, and in effect, messages *ship* and *deliver* will never be enabled. The enactment will be deemed as completed as all the parameters that are needed would be bound.

2.2 Kiko

Kiko is a protocol-based programming model. Kiko's main abstraction is that of a decision maker. Business logic needed for making decisions out of possible

choices resides in a decision maker. Information protocols are used by Kiko as the language for specifying the protocols. By taking over all tasks other than writing the set of decision makers, Kiko enables the developer to focus on the business logic. Figure 1 shows the architecture of a Kiko agent.

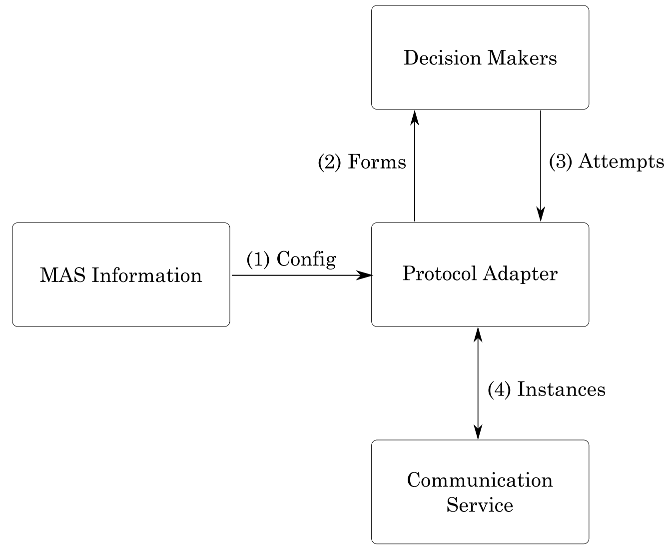

Fig. 1. The Kiko agent architecture [10].

The Multiagent System, Protocol and Decision Makers are authored by the agent developer. Following is an expected workflow of a developer creating a multiagent system:

1. Write an information protocol that defines the roles, messages, parameters and completion criteria for the business process.
2. Write a Python script defining the configuration of the desired multiagent system.
3. Write Python scripts for each involved role by creating an agent using Adapter provided by Kiko.
4. Write decision makers in Python based on the information protocol using the previously written agent.
5. Write a script to start the agents.

The Protocol Adapter on the other hand is provided by Kiko, understands information protocols, and provides an API for plugging in Decision Makers. Let's take a look at a sample flow of information.

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 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 then processed by the protocol adapter as attempts. The protocol adapter then checks the attempts for discrepancies. In case of no discrepancies, 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, as it is sufficient for enacting the information protocols.

The configuration of the MAS defines what agents are present in the system and what roles they play. The configuration also lists different systems that can connect with each other. Each system defines the names of the agents along with, for each agent, the address on which to connect to it. Listing 2 shows an example of how configuration can be set up for MAS based on the protocol in Listing 1. We define one system named `SysName0` that has 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)]
}
```

To initiate the protocol, there are events that can be used by the developer in order to kickstart the system. `InitEvent` is one such event, which is triggered once the agent using it has started. Listing 3 shows one such method authored by the developer that is invoked by the `InitEvent`. For our example, we assume that *Bob* who plays the role of a buyer is requesting a quote for a watch. The message *rfq* is accessible as a form via the enabled argument. Kiko only allows binding the appropriate parameters of the message. The decision maker fails if the form is filled incorrectly.

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)
```

Since our protocol has defined that *rfq* must be sent from a buyer to a seller, Kiko will automatically send this message from Bob to Sally. 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 is able to make 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 set of instances being sent are inconsistent with each other.

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 type 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 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 independent 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 [25] 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.

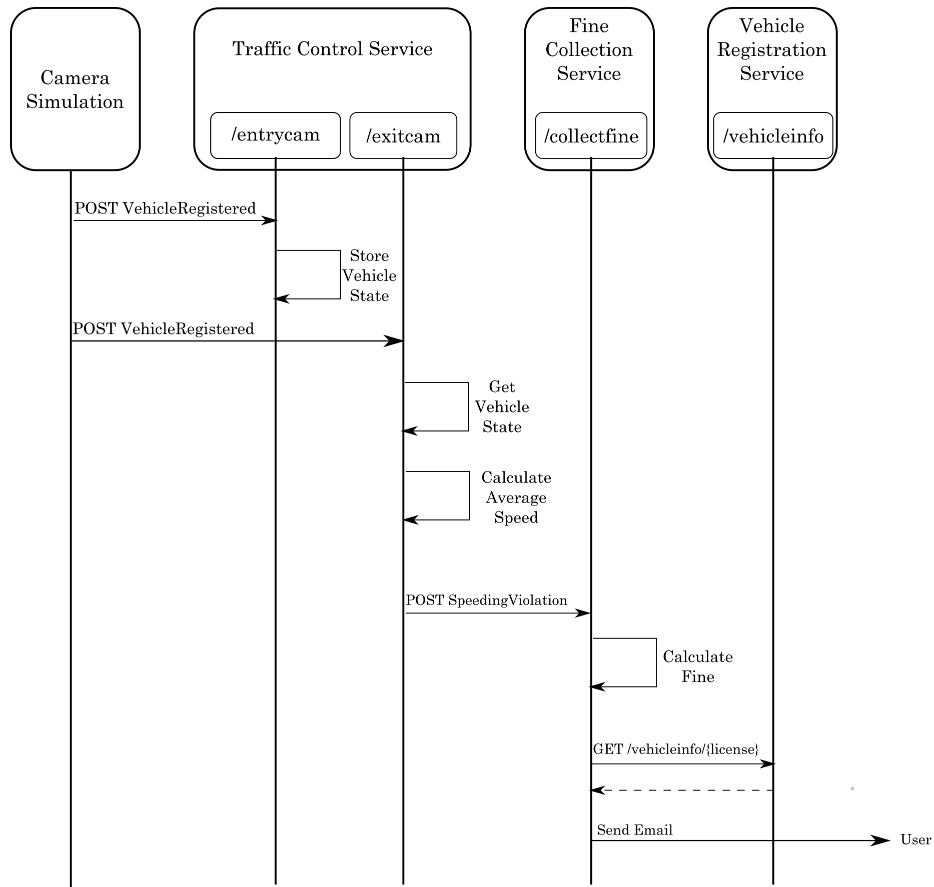

Fig. 2. A UML sequence diagram for the traffic control sample application.

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.

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 a 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.

```
[HttpPost("exitcam")]
public async Task < ActionResult >
    VehicleExitAsync(VehicleRegistered msg, [FromServices]
    DaprClient daprClient) {
  try {
```

```
    // get vehicle state
    var state = await _vehicleStateRepository
        .GetVehicleStateAsync(msg.LicenseNumber);
    if (state ==
      default (VehicleState)) {
      return NotFound();
    }

    // update state
    var exitState = state.Value with {
      ExitTimestamp = msg.Timestamp
    };
    await _vehicleStateRepository
        .SaveVehicleStateAsync(exitState);

    // handle possible speeding violation
    int violation = _speedingViolationCalculator
        .DetermineSpeedingViolationInKmh(
            exitState.EntryTimestamp,
            exitState.ExitTimestamp.Value
        );

    if (violation > 0) {
      var speedingViolation = new SpeedingViolation {
        VehicleId = msg.LicenseNumber,
          RoadId = _roadId,
          ViolationInKmh = violation,
          Timestamp = msg.Timestamp
      };

      // publish speedingviolation (Dapr pubsub)
      await daprClient.PublishEventAsync("pubsub",
          "speedingviolations", speedingViolation);
    }

    return Ok();
  } catch (Exception ex) {
    return StatusCode(500);
  }
}
```

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 key, out exitTS
        key, 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 [6]. Variables that are in uppercase are constants that 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 not necessary 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 for our implementation using Kiko.

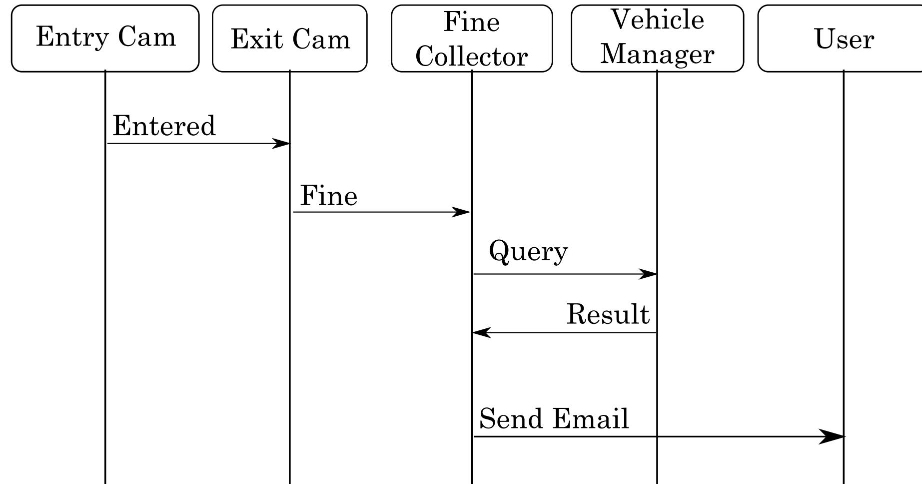

Fig. 3. A UML sequence diagram for our traffic control sample application written using Kiko.

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

4 Evaluation

We would like to evaluate the implementation based on the differences in implementation and the consequent effects. In particular, we see the differences in:

– Internal processing of data.
– Interactions between microservices.
– Safeness of possible interactions between the microservices.
– Developer experience.
– Error handling.

The following points stand out when comparing the Dapr implementation with its Kiko counterpart:

– In the Kiko implementation, **there is no need to store** the exit timestamp of a car to a state, as all observed messages are automatically stored. This enables developers to trace through the messages to understand the flow

of control. In the Dapr implementation, it becomes the developer's duty to store the vehicle information if it is to be used in the future.

- As further invocations are dependent on information passing in the Kiko implementation, we are certain that **variables needed for processing will be known**. In the Dapr implementation, since there is a reliance on the state store for information, the implementation also had to provide additional code to handle that case.
- 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.
- In the Kiko implementation, **the use of information protocols forces developers to build safe and live systems**. A safe system is one that does not have information collision while a live system is one where required information passing is present to ensure that the protocol can be completed. The Dapr implementation on the other hand requires the developer to be cautious while structuring the system. There are no tools present to validate whether a Dapr system is safe and live.
- To get a gist of what the software is attempting to do, we have to take a look at the sequence diagram created by the developers in the Dapr implementation. If the Dapr implementation is updated in the future, the corresponding sequence diagram must also be updated by the developer. On the other hand, Kiko's implementation uses information protocol, which is **the perfect sole document needed to understand the complete system**. Due to the causal structure and information-oriented design of information protocols, it is clear how the flow of the system takes place.

Since the Kiko implementation of the traffic control sample application does not rely on message ordering for processing, it is possible that the event `VehicleExit` is received prior to the reception of the *Entered* message. In this case, as the *Fine* message would never be activated, it would simply ignore the event. There is no explicit processing added to check for the case of the *Entered* message being received after the `VehicleExit`. 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. Similarly, in case Kiko receives a message that is not appropriate for the current state of the agent, it is ignored and discarded.

It is also easier to suspend and continue jobs that use information protocols as every message can be thought of as a new layer that is to be built upon. In our sample application, a power failure that restarts all microservices would support continuing the processing of an enactment as long as the local store has the correct messages. Duplicate messages are handled by Kiko and do not prevent processing. Due to the causal nature of information protocols, it is simple to program continuation of these processes. An application written in Dapr would

fail to do this as the system does not understand partial completion unless explicitly programmed by the developer.

Dapr still relies on the request-response style of programming that suffers from the need of receiving messages in the order they were sent to ensure coherent processing. The Kiko implementation uses lossy, unordered communication protocol UDP, to send messages rather than use a request-response style.

Based on this evaluation, we can conclude that the Kiko implementation provides better developer experience, and the causal structure of the protocol leads to a safe and live program that is verified by other tools.

5 Discussion

Using the microservice architecture also requires a fair amount of knowledge dealing with the deployment of different services. A dedicated DevOps team was found to be necessary for software that followed microservice architecture [21]. With only about 10% of respondents claiming to be a DevOps specialist in the 2022 Developer Survey [20] 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 [12]. Integration of Kiko along 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 [11].

Multiagent systems need to provide an account of what happened during an abnormal situation [3, 5]. 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 [14] and this idea is supported in information protocols in its causal nature and its ability to use lossy, unordered protocols such as UDP. A lesson learned in the explorative study of promises and challenges in microservices [24] was 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 production environment. Timeouts within a microservice system is a problem that is remedied by using a circuit breaker [23] 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 [4]. Verification at design time [2] 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 ways in which decision makers may be invoked when a specified set of events and/or 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

Thanks to the EPSRC (grant EP/N027965/1) and the NSF (grant IIS-1908374) for support.

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., 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`
3. Baldoni, M., Baroglio, C., Micalizio, R., Tedeschi, S.: Accountability in multi-agent 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`
4. 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)

5. 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
6. Christie, S.: Kiko, https://gitlab.com/masr/bspl/-/tree/kiko/, accessed 15 February 2023
7. 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
8. 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
9. 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
10. 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)
11. 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)
12. 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)
13. Dapr: Dapr – Distributed Application Runtime (2019), https://dapr.io/, accessed 14 February 2023
14. 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
15. 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
16. PwC: Cloud business survey, https://www.pwc.com/us/en/tech-effect/cloud/cloud-business-survey.html, accessed 14 February 2023
17. Richardson, C.: Monolithic architecture pattern, https://microservices.io/patterns/monolithic.html, accessed 8 February 2023
18. 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
19. 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
20. Stack Overflow: Stack Overflow 2022 Developer Survey, https://survey.stackoverflow.co/2022/, accessed 14 February 2023

21. 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
22. Thönes, J.: Microservices. IEEE Software **32**(1), 116–116 (2015). https://doi.org/10.1109/MS.2015.11
23. 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
24. 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
25. van Wijk, E., Molenkamp, S., Hompus, M., Kordowski, A.: Dapr traffic control sample, https://github.com/EdwinVW/dapr-traffic-control, accessed 15 February 2023