Protocol-Based Engineering of Microservices

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

[1]North Carolina State University

[2]Unaffiliated

[3]Lancaster University

EMAS, May 2023

Table of Contents

Table of Contents

Motivation

Can microservices benefit from MAS abstractions?

- Microservices: Industry paradigm for decentralized, loosely coupled software
  - Supported by programming models such as Dapr
- Challenge: Coordination between components
  - Multiagent systems (MAS) engineering: coordination via protocols

Contributions and Conclusion

- We contrast a conventional Dapr implementation with one based on Kiko (AAMAS 2023), a programming model for information protocols
- Information protocols-based engineering highly promising for microservices

Table of Contents

Dapr

Microsoft-originated, now community-driven, open source

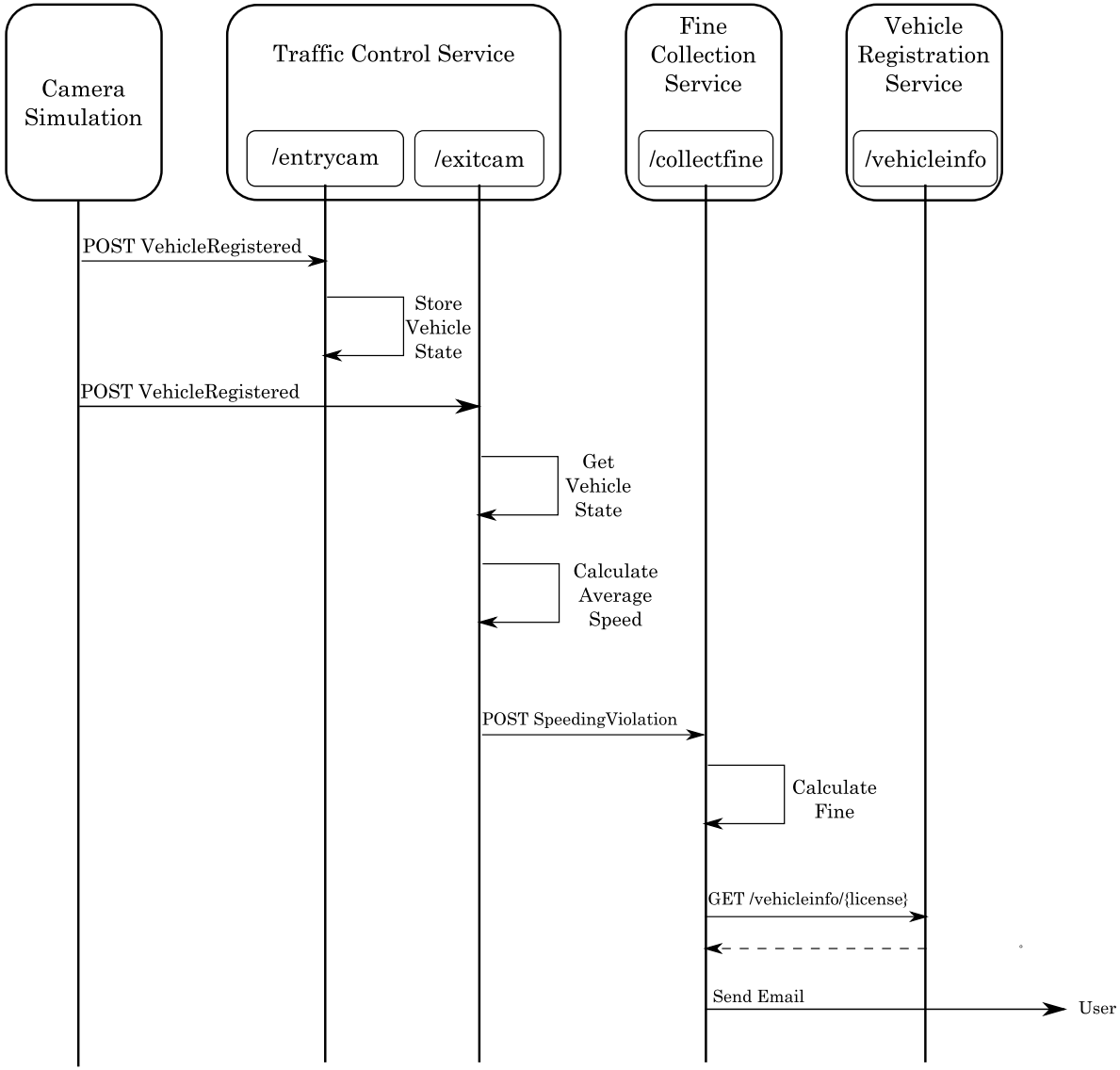
- Event-driven runtime
- Promises resilient interoperable microservices
- Provides reusable building blocks for applications
  - State store: Used as a database
  - PubSub brokers: Used as a message queue
  - Bindings & triggers: Used to communicate with external services.

Traffic Control Application

Inspired by the speeding-camera setup present on some Dutch highways

- Camera is simulated
- Traffic Control Service
  - `/entrycam`: when vehicle enters section
  - `/exitcam`: when vehicle leaves section
- Fine Collection Service
- Vehicle Registration Service

Interactions in the Application (Notice Shared State)



Exitcam Endpoint

```
public async Task RegisterExitAsync(VehicleRegistered msg)
{
    try
    {
        Logger.LogInformation($"EXIT detected in lane {msg.Lane} at " +
            $"{msg.Timestamp.ToString("hh:mm:ss")} " +
            $"of vehicle with license-number {msg.LicenseNumber}.");

        // remove lost vehicle timer
        await UnregisterReminderAsync("VehicleLost");

        // get vehicle state
        var vehicleState = await this.StateManager.GetStateAsync<VehicleState>("VehicleState");
        vehicleState = vehicleState with { ExitTimestamp = msg.Timestamp };
        await this.StateManager.SetStateAsync("VehicleState", vehicleState);

        // handle possible speeding violation
        int violation = _speedingViolationCalculator.DetermineSpeedingViolationInKmh(
            vehicleState.EntryTimestamp, vehicleState.ExitTimestamp.Value);
        if (violation > 0)
        {
            Logger.LogInformation($"Speeding violation detected ({violation} KMh) of vehicle " +
                $"with license-number {vehicleState.LicenseNumber}.");

            var speedingViolation = new SpeedingViolation
            {
                VehicleId = msg.LicenseNumber,
                RoadId = _roadId,
                ViolationInKmh = violation,
                Timestamp = msg.Timestamp
            };

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

Table of Contents

Developer Workflow

1. Specify an information protocol
2. Implement each agent in Python by
   - Writing decision makers that emit messages
   - Configuring the agent's with information about the MAS and the network addresses of other agents in the MAS
3. Launch the agents

Protocol

```
TrafficControl {
    roles EntryCam, ExitCam, FineCollector, VehicleMngr

    parameters out regID key, out entryTS key, out exitTS key, out
        email

    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]
}
```

UML Sequence Diagram

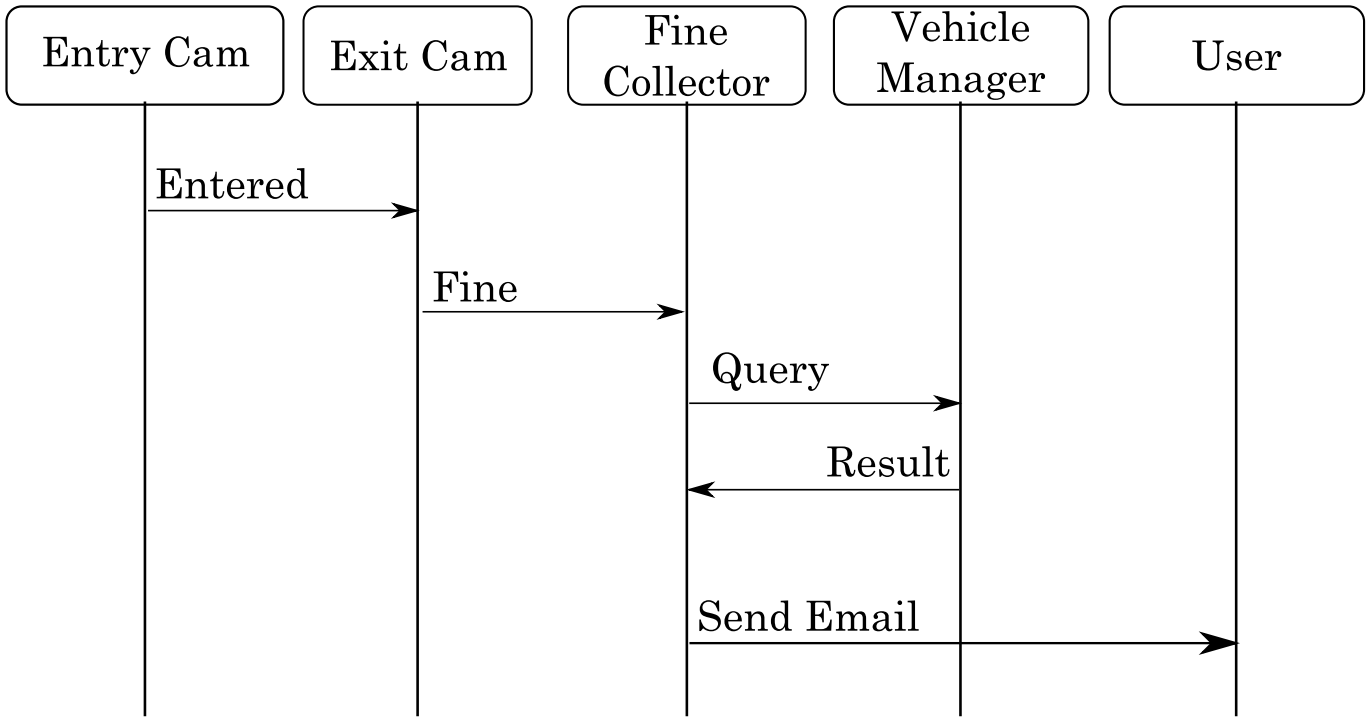


Figure: A UML sequence diagram for our traffic control sample application written using Kiko.

Kiko

- Kiko is a protocol-based programming model.
- Kiko's main abstraction is of a decision maker.

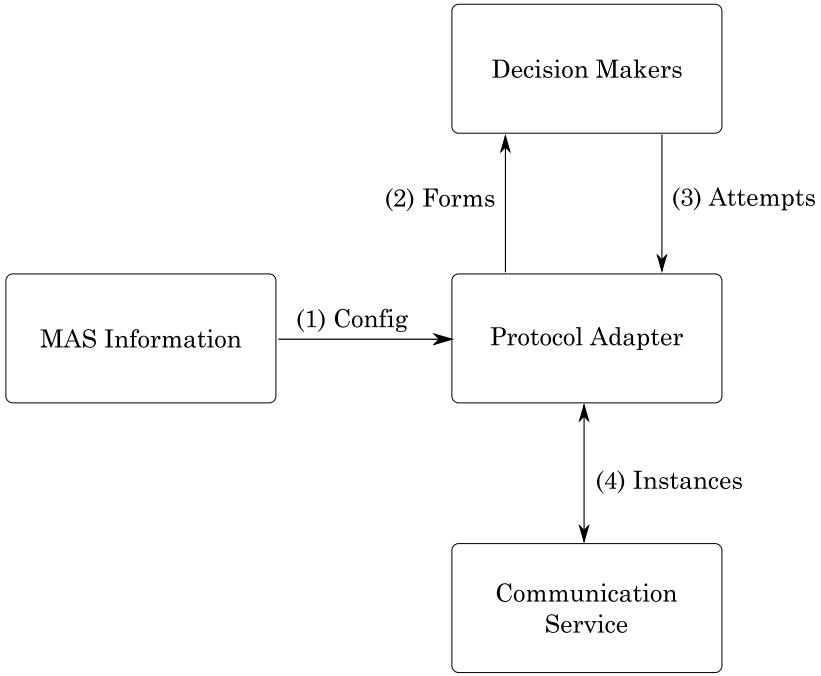


Figure: The Kiko agent architecture

Forms as Causally-Enabled Communications

Message schemas with ⌜in⌝ parameters filled

```
ExitCam -> FineCollector: Fine[in regID, in entryTS, out
    exitTS, out avgSpeed]
```

Entered(1, 1000)
Entered(5, 1100)
Fine(1, 1000, 1001, 100mph)

ExitCam's local state

Fine(5, 1100, exitTS, avgSpeed)

ExitCam's Forms

Exit Camera Agent

```
@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
```

Table of Contents

Kiko Advantages

- Kiko yields truly decentralized implementations without shared state
- Kiko code is simpler
  - Logging code reduced because Kiko logs all messages
  - Error (no exit for entry) handling off decision making path
  - Give semi-populated forms, reducing code and the possibility for errors
- The information protocol serves as a formal document of system architecture
  - Can be verified for safety and liveness
  - Kiko guarantees agents comply with the protocol
- Communication in Dapr is either request-response or message queue-based. The Kiko implementation uses asynchronous messaging via UDP.

Future Work

- Push old messages in the log to archival storage for efficiency
- Catalog popular microservices patterns and map them to Kiko
- Build serverless implementation of Kiko
- Make Kiko available as a Dapr component
- Develop tooling and IDE support geared toward information protocols