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.

Delivering Multi-Agent MicroServices using CArtAgO

Eoin O'Neill[1], David Lillis[1][1111−2222−3333−4444], Gregory M. P. O'Hare[1][2222−−3333−4444−5555], and Rem W Collier[1][3333−−4444−5555−6666]

School of Computer Science, University College Dublin, Dublin, Ireland
eoin.o-neill.3@ucdconnect.ie
{david.lillis,gregory.ohare,rem.collier}@ucd.ie

Abstract. This paper describes an agent programming language agnostic implementation of the Multi-Agent MicroServices (MAMS) model - an approach to integrating agents within microservices based architectures. In this model, agents, deployed within microservices, expose aspects of their state as virtual resources that are externally accessible using REpresentational State Transfer (REST). Virtual resources are implemented as CArtAgO artifacts, exposing its state to the agent as a set of observable properties. Coupled with a set of artifact operations, this enables the agent to monitor and manage its own resources. In the paper, we formally model our approach, defining *passive* and *active* resource management strategies, and illustrate its use within a worked example.

Keywords: First keyword · Second keyword · Another keyword.

1 Introduction

This paper builds on previous work that has introduced the *Multi-Agent MicroServices (MAMS)* model [31][21]; a model that promotes a view of agents as hypermedia entities whose body includes a set of virtual resources that can be interacted with using REpresentational State Transfer (REST) [9] and can be deployed as microservices. Overall, the work has three main objectives: to facilitate the seamless deployment of Multi-Agent Systems (MAS) within microservices ecosystems; to exploit modern industry tools to enhance the deployment of MAS; and ultimately, to enable the development of an emerging class of systems known as *Hypermedia MAS* [3][4].

The specific focus of this paper is to improve on the approach described in [31] by proposing an agent-programming language independent approach based on CArtAgO [25] and to introduce support for hypermedia links through the use the Hypertext Application Language (HAL) [14] as described in [21]. To achieve this, Section 3 describes the refined MAMS model; Section 4 introduces the suite of CArtAgO actifacts developed to implmement the model; and Section 6 illustrates its use through a worked example. Finally, Section 8 presents some concluding remarks.

2 Related Work

There has been a significant amount of research into the integration of agents and services. [4] provides a good historical perspective on this. [11] is an excellent overview of agent-based service-oriented computing, with a focus on Web Services technologies. [22] is an excellent recent survey of agent-based cloud computing applications that is heavily focused on agent-based service-oriented computing (in the cloud). Much of it tackles the relationship between agents and services from a more traditional perspective. In this paper, the objective is to focus more closely on the relationship between agents and microservices - a architecture style linked to service-oriented computing that promotes a more decentralised approach to software development.

Microservices are increasingly seen as an important innovator in software design. They champion the decomposition of monolithic systems into loosely-coupled networks of services [30] that are necessary to deliver internet-scale applications [8]. This has the effect of reducing the complexity of many of the components, but comes at the cost of increasing the complexity of deployment [29]. However, this challenge has been met through the rise of DevOps [2] and Continuous Software Engineering methods [20].

The rise of microservices presents an opportunity for Multi-Agent Systems (MAS) research. As is illustrated in [31], there is a strong affinity between the principles of microservices and MAS that can be exploited to deliver innovations, both in terms of the use of MAS technologies with microservices and the use of microservices technologies with MAS. This affinity is reinforced in [27], which argues that microservices can be used to facilitate agility of development for agent-based Internet of Things (IoT) systems. This view is further reinforced in [16], which argues that microservices-based IoT systems can be modelled as agents, and in [15], which presents a multi-agent trust model for IoT.

While not directly referencing microservices, [18] argues for a new "Agents as a Service" paradigm that would enable a new generation of agile services founded on the MAS models and techniques. Similarly, [31] argued for the emergence of an "Organisation as a Service" paradigm in which MAS implementations of organisational models are implemented and deployed as microservices that can be utilised by other non-agent-based microservices.

Key to realising this vision of agents and microservices is the need for dedicated programming tools and frameworks that help to simplify the development process. To date, there have been two main attempts to achieve this. [32] introduces CAOPLE: a Caste-centric Agent-Oriented Programming Language and Environment for programming microservices. Conversely, [31] presents an extension to ASTRA [5] (a variant of AgentSpeak(L) [23]) that supports the implementation of microservices.

3 Multi-Agent Micro-Services

The concept of Multi-Agent Micro-Services (MAMS) was originally introduced in [31]. The paper argues that microservices share many common traits with

Multi-Agent Systems (MAS), to the extent that both approaches can be broadly characterised as being concerned with the creation of loosely-coupled distributed systems comprised of small independent (autonomous) components with internal state. Of course, there are also many differences between the two approaches, not least the incorporation of practical reasoning, but this commonality suggests that we are beginning to see the emergence of approaches within industry that are, at least, compatible with the MAS perspective.

As mentioned in the introduction, the ultimate goal of MAMS is to allow agents to be deployed as entities, that co-exist with other agents and resources in a *hypermedia space*. This space encompass all resources that can be: addressed using a Uniform Resource Identifier (URI); accessed using the HyperText Transfer Protocol (HTTP); and connected through a network of hyperlinks. Through these aspects, agents become not only identifiable and discoverable, but also observable. That is, the body of an agent can be directly observed and interacted with through the use of appropriate HTTP requests.

This notion of observability can lead to direct benefits in terms of enabling emergent behaviour. To illustrate this, consider a scenario in which a person is sitting in a public space and overhears a conversation between other people in that space. By listening to the conversation, the person is able to build not only models of the beliefs and goals of the other people, but also the protocols/rules that underpin the conversation. At some point the person may interject into the conversation simply by applying the learnt protocols/rules. With MAMS, this type of behaviour could be replicated by modelling inboxes as virtual resources that return a filtered view of the agents conversation history upon receipt of a GET request. Performing the GET request is the agent equivalent of listening in to the conversation of the other agent. Upon receipt of the conversation history, it could mine the messages to not only understand: what beliefs the other agent has, what services it provides, and what protocols it uses for that interaction; but also the URIs of the other agents that it has interacted with.

3.1 Basic MAMS Model

At its core, MAMS adopts the view of a microservice as a container for one or more agents. Agents may be internal (private) to the container or external (public). Public agents are associated with a Uniform Resource Identifier (URI) based on a combination of the hostname and port of the service plus the name of the agent. They are also associated with a hypermedia body that is constructed from a set of virtual resources. Private agents have standard string based identifiers and no hypermedia body. Both types of agent should be implemented using a common agent programming language or framework. A microservice is not considered a MAMS service if there are no public agents.

Figure 1 presents a sketch of a standard layered microservices architecture, access to which is mediated by an API Gateway, a common microservices design pattern[1] that employs a front-facing service that acts as a single point of

[1] https://microservices.io/patterns/apigateway.html

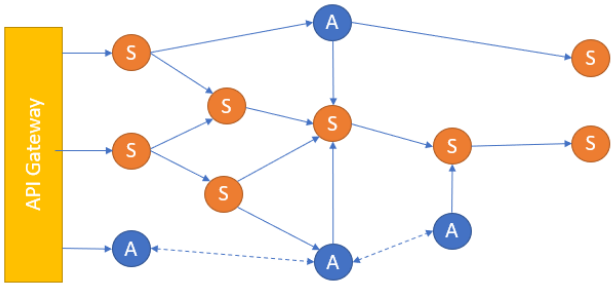

Fig. 1. Agent/Service Integration

entry to the layered architecture. In the context of this architecture, one of the goals of MAMS is to facilitate interaction between agent-based (A) and non agent-based (S) microservices (i.e. agent-agent, agent-service, service-agent and service-service). Agent-service interaction is achieved by giving agents the ability to submit HTTP requests and process HTTP Responses. Service-agent interaction is achieved by exposing the virtual resources through REST using URIs - based on the associated agents URI - providing a HTTP-based interface that does not require knowledge of agent concepts.

Agent-agent interaction, using an Agent Communication Language (ACL) or equivalent, can also be realised through virtual resources. Specifically, each agents inbox can be modelled as a virtual resource. Sending a message to an agent is reduced to submitting a HTTP POST request to the receiver agents inbox URI, with the content of the request being the message. This approach is demonstrated later in Section 4.4. It is useful to note that FIPA's HTTP message transport service specification [19] works in a similar, if more convoluted, way: the sender agent passes the message to a message transport service which wraps the message in an envelope. It then POSTS the wrapped message to an Agent Communication Channel hosted on the receiver agents platform which unwraps the message and delivers it to the relevant agent.

Another form of agent-agent communication that is supported by MAMS is through non-ACL based virtual resources. Each agent is also able to interact with other agents using the same model that is used for agent-service interaction. While this can be used in place of ACL based interaction, the more interesting scenario is where the virtual resources represent abstractions of the agent state, such as, public beliefs and goals, lists of acquaintances, or services offered. Such resources would be publicly accessible and, as such, discoverable by other agents. This posits a view where agents could directly observe the behaviour of other agents, explore the acquaintance networks of their own acquaintances, and potentially seek out other agents that provide the services that they need access to. Hyperlinks are essential to achieving this vision. While hyperlinks could be used externally to identify virtual resources, relevant links were not included in the resource representations returned by HTTP requests. [21] extended the basic

MAMS model in include such hyperlinks based on the Hypertext Application Language (HAL). This extended model is described briefly in the next section.

3.2 Extending MAMS with HAL

The introduction of a resource representation that includes hyperlinks is a key step in achieving the vision of agents as hypermedia entities. There are many potential technologies for use in this area, and [17] presents a good summary of those in the context of the Web of Things. In [21], we selected Hypertext Application Language (HAL) [14] for adoption with MAMS. While HAL is not an IEFT standard, it is one of the simplest linked data models to have been proposed and is relatively easy to implement, and is currently in use in multiple projects[2].

```
{
    "_links": {
        "self": { "href": "/api/books/1234" }
    }
    "id": 1234,
    "title": "Hitchhiker's Guide to the Galaxy",
    "author": "Adams, Douglas"
}
```

Fig. 2. Example HAL resource representation (from [10])

HAL augments JSON representations with additional keys prefixed by an underscore (_). The `_links` key is used to define a set of named hypermedia links relevant to the resource being represented. For example, the JSON in Figure 2, represents a book resource. The `self` link, is the URI of the representation itself. Additional links can added that define operations specific to the resource (e.g. a library system may add a link to the loan resource for that book).

A weakness of HAL is that the semantics associated with the links is application dependent. Using HAL requires the definition of what valid links can be used for each resource. In response to this, best practice for use use of REST in industry was reviewed. This highlighted that many REST APIs focus on two styles of resource: individual items and lists of items [26] that were manipulated through the mapping of HTTP verbs to CRUD operations. Based on this, it was decided that a generalised implementation of these resource types would be developed based on this best practice.

Table 1 contains a summary of the standard set of HTTP verbs that are associated with these resource types and their associated behaviours. For example, as can be seen in this table, it is increasingly common for individual items (singleton resources) to support retrieval of the state using a GET request and a

[2] https://github.com/mikekelly/hal_specification/wiki/APIs

Resource Type	URI	POST	GET	PUT/ PATCH	DELETE
Item	`/{id}`	n/a	Get the item	Update the item	n/a
List	`/{list_name}`	Add to the list	Get entire list of items	n/a	n/a
ListItem	`/{list_name}/{id}`	n/a	Get the item	Update the item	Remove item from list

Table 1. Core resource types and key HTTP verb mappings

partial/full update using PATCH/PUT. POST operations are typically not permitted because they are creation-oriented (which does not apply to a singleton). Similarly, DELETE operations are typically not supported because there is no way to recreate the resource once it is deleted.

The choices described above represent just one possible resource implementation strategy for MAMS. It was made in an effort to facilitate exploration of the MAMS model. Our long-term goal is to explore the use of JSON-LD [28] due to its use of the Resource Definition Framework (RDF) as a schema [6].

4 An Artifact-based Framework for building MAMS agents

To illustrate the MAMS model, a prototype implementation has been developed. When designing the prototype, two potential approaches were discussed: creating a bespoke implementation from first principles, or to adapting an existing framework. In this paper, the latter approach was preferred because our goal is to provide an agent language agnostic implementation. To this end, the CArtAgO framework [25] was chose because it was felt that virtual resources can be modelled as artifacts and because CArtAgO is an established and tested technology that is integrated with multiple established agent programming languages. This has allowed us to focus on the model rather than lower level integration issues.

A key difference between MAMS and the CArtAgO approach is that artifacts combine observable properties (state) and operations (behaviour) while virtual resources support only state. In our implementation, the state of a virtual resource is modelled as observable properties. Operations are provided to enable the agent to manipulate the resource (e.g. updating an observable property, linking the artifact, ..) in a way that is compliant with the MAMS model. A limitation of using CArtAgO is that virtual resources are private, but artifacts are designed to be shared. This means that it is possible to misuse our implementation and an agent within the same microservice could gain access to another agents virtual resources. We ignore this issue.

To implement the MAMS model, a number of artifacts have been developed that represent key concepts. A high-level view of our approach is illustrated in

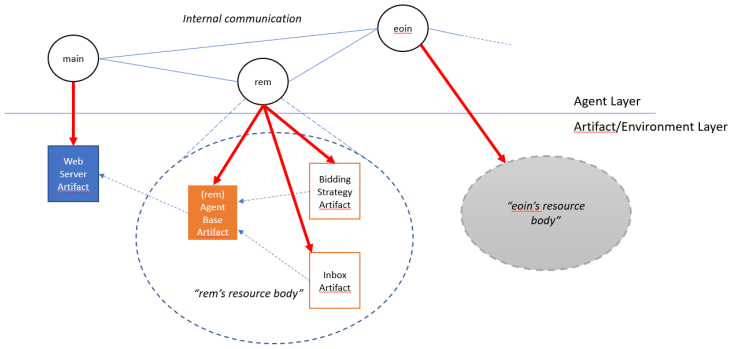

Fig. 3. Modelling a RESTful agent body as artifacts

Figure 3, where each agent is associated with a hypermedia body, consisting of a set of CArtAgO artifacts that model the virtual resources of each agent. A `base` artifact is provided as a shared base to which each `resource` artifact is linked and this in turn is linked to a shared `webserver` artifact that exposes the resources over HTTP. The webserver artifact is implemented using the Netty: an asynchronous Java-based event-driven network application framework for high performance protocol servers [3].

Modelling resources explicitly as artifacts allows for clearly-defined semantics that includes a description of how each HTTP verb will affect the state of the artifact modelling the associated resource. It also specifies the interface between the agent and the resource, which is defined in terms of the operation, observable property, and signal concepts of CArtAgO. This paper describes two approaches to implementing virtual resources as artifacts: a passive resource management model (Section 4.2), and an active resource management model (Section 4.3). However, before discussing these approaches, Section 4.1 describes how artifacts are used to implement virtual resources.

Two additional artifacts are created when a MAMS microservice is started. The `restclient` artifact implements a REST client that can be used to perform REST API calls. For example, the `postRequest` operation takes a URI and string representation of a json body as input and generates a response code and string content. Similar operations exist for GET, PUT and DELETE. The `comms` artifact provides support for sending FIPA-ACL style messages to other MAMS agents. More information on this is provided in Section 4.4.

4.1 Implementing Virtual Resources as Artifacts

Figure 4 illustrates how MAMS exposes artifact-based virtual resources on the web and the relationship between an agent and the associated set of artifacts that implement those resources. Each artifact created by the agent is linked to

[3] https://netty.io/

another artifact creating a back channel through which incoming HTTP requests are routed to the relevant artifact. The back channel consists of a set of *handlers* that implement the routing behaviour. Collectively, the set of handlers form a tree structure rooted at the `base` artifact. Each handler is associated with a single artifact and each path from the root to a handler represents the URI of a virtual resource of the agent.

Listing 1 contains some pseudo code for creating the body of an agent. As can be seen, the agent starts by retrieving a reference to the `webserver` artifact. Once it has this, a `base` artifact is created. This artifact is given a name of the form `base-<aid>` and is then linked to the `webserver` artifact. The agent focuses on the newly created artifact so that it can get updates on observable properties and signals. Finally, a `createRoute()` operation is executed on the newly created `base` which creates the associated handler and links it to the `webserver` artifacts handler. The `base` artifact id has been chosen to ensure that all artifact names are unique.

```
1 lookupArtifact("webserver", id)
2 makeArtifact("base-<aid>", "mams.BaseArtifact",[<aid>], id2)
3 linkArtifacts(id2, "out-1", id)
4 focus(id2)
5 createRoute()[id2]
```

Listing 1: Pseudo Code for Creating the body of a MAMS Agent

To support the model described in Table 1, three additional types of artifact are required: `item`, `list` and `listitem`. However, the exact form that each of these artifacts take depends on whether we are using a passive or active management model.

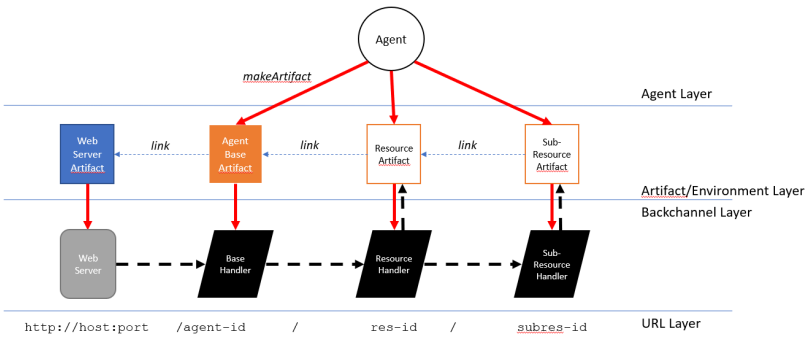

Fig. 4. Use of CArtAgO artifacts for linking RESTful resources to agents

Section 4 defined the representation of a `item` resource as the set of observable properties associated with it. Receipt of a request for a representation of that resource involves transforming the observable properties into that relevant representation format. For the model described in section 3.2, we use a Java object an an intermediary format that is transformed to/from JSON using the Jackson API[4]. For GET requests, the `_links` field is appended to the resulting JSON object based on the linkages that exist between artifacts (those that are used to form the backchannel). For POST, PUT and PATCH requests, the JSON is transformed into a Java object whose fields correspond to the observable properties of the artifact and whose values are used to update those propertied. To facilitate this, each `item` or `list` artifact is associated (at creation time) with a Java class that defines what type of object is to be used for the intermediary format. On creation either default values are used it initialise observable properties or an instance of the class is passed providing the initial values.

4.2 Passive Resource Management

In the passive model, agents are not responsible for enforcing the changes associated with any HTTP requests received. They simply act in response to resource changes. How the resource is updated depends on an associated set of semantics which is loosely described in Table 1.

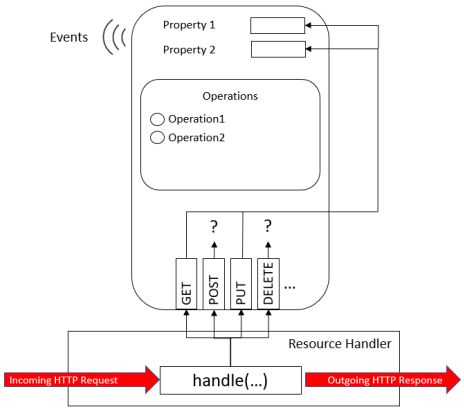

Fig. 5. Schematic of a Passively Managed Resource

As the artifact receives each request, depending on the HTTP verb used, the agent receives a CArtAgO signal indicating the nature of the update that was applied. This allows the agent to act in response to expected changes in the

[4] https://github.com/FasterXML/jackson

resources, but does not affect the speed by which the response returned to the system making the request. Additionally, the agent is also able to make changes to the state of the resources through a suite of internal operations.

This idea allows rapid interaction between the resource and the entity making the request, while maintaining that the agent is still informed about the state of each resource. A key factor of this method is the fact that although the agent may have control over the resource, the resource is open to the world as an endpoint. This permits any service (or agent) to make a request and receive a timely response from this entity, something that may not be possible when you introduce the mentalistic aspect of deliberation that is associated with agents.

In terms of usage, this type of resource model seems suited either to closed systems where trust is not an issue and all resource changes follow expected patterns of behaviour or to open systems where the manipulation of resources is a desirable aspect.

4.3 Active Resource Management

For active management, each artifact also has a set of HTTP verbs that it can handle based on Table 1. In contrast, however, the agent is no placed in control of the response given by any artifacts under its stewardship. This is illustrated in Figure 6.

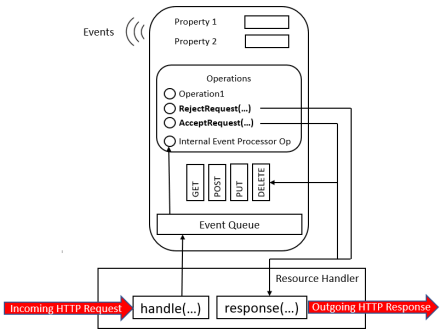

Fig. 6. Schematic of an Actively Managed Resource

Once a valid HTTP request is made of an artifact, a CArtAgO signal is generated based on the type of HTTP verb passed to the agent. For GET and DELETE requests, the request body is ignored. Conversely, the body is included for POST, PUT and PATCH requests. This event is passed to the agent which then deliberates to decide on the correct response.

If deemed acceptable, the agent executes the "accept" operation on the artifact. The request is then be removed from the event queue and processed. A

response detailing that the request made was handled correctly would be issued. In the case where the request was rejected by the agent, the "refuse" operation would be invoked, issuing a response that the request was denied.

With regard to a use case, this scenario can be utilised when dealing with a resource that is highly constrained and only wants to accept requests of a given standard/type. This then lends itself to Quality of Service (QoS) based systems, as it allows the system to guarantee certain criteria with regard to the manipulation of resources.

4.4 FIPA-ACL Based Interaction

ACL-based communication between MAMS services is supported through the creation of a custom `inbox` virtual resource. This resource is somewhat similar to a standard `list` resource, with the exception that it only accepts POST requests. This resource is designed to be used in tandem with the `comms` artifact, which includes an operation for sending messages to MAMS agents via a POST request. The content of the message is submitted in the form of a JSON string. It is left to the developer to decide how to generate this content.

In its current form, the FIPA Message class that models messages only includes the: sender, receiver, performative, language and content fields. The content itself is converted into a JSON string that is transmitted as the body of the POST. A signal is generated by the `inbox` resource for each message received. The content of this signal the performative, the sender URI, and a string representation of transmitted content. Again, conversion of this JSON into a more useful form is left to the developer. The current prototype is currently released as part of the ASTRA-MAMS integration, which is described in more detail next in Section 5 and comes with built in support for converting functional terms into JSON and vice versa. It should be noted that the artifacts describe here do not map onto the model defined in section 3.2 but are purpose-built to facilitate FIPA-based interaction.

5 Integration with ASTRA

To further explore our MAMS model, we have integrated our CArtAgO based solution with the ASTRA agent programming language [5][7]. ASTRA is an implementation of AgentSpeak(ER)[24] a recent evolution of AgentSpeaK(L)[23].

All of the source code for MAMS and for the ASTRA integration with MAMS open source and available to download from Gitlab [5]. This includes:

- `mams-cartago-core` package: contains the `webserver`, `restclient`, and `base` artifacts; support for handlers and a basic web server.
- `mams-cartago-hal` package: contains the implementation the item, list and itemlist artifacts together with support for Java classes as schema.

[5] https://gitlab.com/mams-ucd

```
1  agent MAMSAgent {
2    rule +!setup() {
3      cartago.startService();cartago.link();
4      cartago.makeArtifact("webserver", "mams.artifacts.WebServerArtifact",
5              cartago.params([9000]), cartago.ArtifactId id);
6      +artifact("webserver", "webserver", id);
7      cartago.makeArtifact("restclient", "mams.artifacts.RESTArtifact",
8              cartago.params([]), cartago.ArtifactId id2);
9      +artifact("restclient", "restclient", id2);
10     cartago.makeArtifact("comms", "fipa.artifact.Comms",
11             cartago.params([]), cartago.ArtifactId id3);
12     +artifact("comms", "comms", id3);
13   }
14   inference have(string name) :-
15                 artifact(name, string qname, ArtifactId id);
16   rule +!init() {
17     cartago.link();!have("webserver");!have("restclient");
18   }
19   rule +!have(string name) : ~have(name) {
20     cartago.lookupArtifact(name, cartago.ArtifactId id);
21     +artifact(name, name, id);
22   }
23   rule +!created("base") : ~created("base") &
24         artifact("webserver", string qualifiedName, ArtifactId id2) {
25     string baseName = S.name()+"-base";
26     cartago.makeArtifact(baseName, "mams.artifacts.BaseArtifact",
27         cartago.params([S.name()]), cartago.ArtifactId id);
28     cartago.linkArtifacts(id, "out-1", id2);
29     cartago.focus(id);cartago.operation(id, createRoute());
30     +artifact("base", baseName, id);
31   }
32 }
```

Listing 2: Part of the mams.MAMSAgent program

- `mams-astra-hal` package: contains the integration of ASTRA and the MAMS +HAL model and the prototype FIPA-ACL based communication model.
- `examples`: a Gitlab group containing a set of sample programs including an example of interaction with a Spring Boot Service. All examples are implemented as Maven projects. Details of how to run ASTRA using Maven can be found on the ASTRA website[6].

The main ASTRA code for creating a MAMS Agent is implemented in the `MAMSAgent` class. Partial code for this class is shown in Listing 2. The `+!setup()` rule on lines 2-13 is invoked only once by the first agent to be created. This plan basically configures the MAMS service, creating all the default artifacts. In

[6] http://www.astralanguage.com

```
1  agent MAMSAgent {
2    module mams.HALConverter hal;
3
4    rule $cartago.signal(string sa,
5              message(string perf, string sender, string content)) {
6      !signal_message(perf, sender, hal.toRawFunct("content", content));
7    }
8    rule +!signal_message(string performative,
9              string sender, content(funct content)) {
10     !message(performative, sender, content);
11   }
12   rule +!transmit(string perf, string receiver, funct content)
13           : artifact("comms", string qname, ArtifactId id) {
14     !itemProperty("base", "uri", funct agentUri);
15     cartago.operation(id, transmit(perf, F.valueAsString(agentUri, 0),
16             receiver, hal.toJsonString(content(content))));
17   }
18 }
```

Listing 3: FIPA ACL Code from mams.MAMSAgent class

contrast, the `+!init()` rule on lines 16-18 are to be used by all other MAMS agents. The associated goal is used to link the agent to the already created artifacts. Once the `!init()` goal has been achieved, the agent is able to create the base artifact using the rule on lines 23-31.

The snippet of code in Listing 3 relates to the support for FIPA ACL based communication. The module on line 2 includes support for for converting functional term into JSON and vice-versa. The `+!transmit()` rule on lines 12-17 implement support for sending messages. This is matched by the rule on lines 4-7 which intercepts the raw CArtAgO signal relating to an incoming message. The rule invokes a chain of subgoals that results in the conversion of the raw content of the message back into a form that corresponds more closely to a normal ASTRA message event. The `!message(...)` goal generated on line 10 should be could by the implementing agent to handle receipt of specific FIPA messages.

6 Illustration

To demonstrate our approach, a version of the Vickrey Auction example presented in [31] has been built using the framework described in Section 5. The resultant code base is quite different because the original approach which mixed code for handling HTTP requests and responses with code for implementing the auctions. In contrast, the code in our approach is more focused on implementing the auctions.

The implemented system exposes a set of virtual resources that are linked to specific agents within the implementation. As shown in Figure 7, the `Manager` agent is associated with the `/clients` and the `/items` resources and the `Bidder`

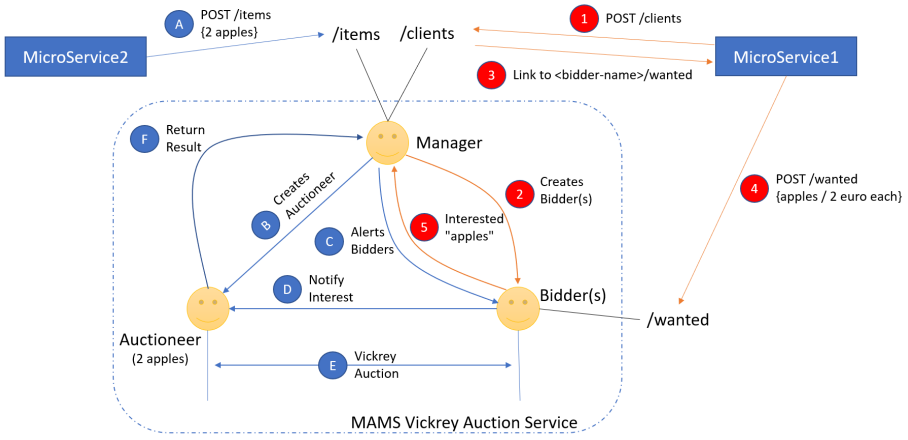

Fig. 7. Vickrey Auction Implementation (taken from [31])

agents, which are created by the `Manager`, are each responsible for their own `/wanted` resource.

The `mams.PassiveMAMSAgent` agent program provides plans to support the creation of passively-managed artifacts. Listing 4 shows a plan that can be used to create a `list` resource. Below this, a second piece of code from the `Manager` agent program illustrates how to use this to create a list of clients. It also demonstrates how CArtAgO signals are used to alert the agent to the creation of new items. A templating mechanism is provided that uses Java classes (here the `auction.Client` class) as a schema for items.

7 Discussion

One of the main benefits of the approach presented in this paper is that it standardises how to build MAMS-based applications. This has led to a number of improvements compared against the initial implementation of MAMS as described in [31]:

- *Explicit Modelling of Resources:* The original MAMS model maintained an implicit model of resources whose state was represented within the agents' beliefs. The approach advocated in this paper models resources explicitly. A key benefit of this has been the ability to define explicit resource types (see Table 1), with associated semantics for valid HTTP Requests, that are encoded within the the resource model.
- *Support for Extensibility:* The implementation of resources is designed to be extensible and permit the addition of other resource types as is necessary. This is essential as it permits the development of bespoke resource models

```
agent PassiveMAMSAgent {
  rule +!listResource(string name, string cls)
        : ~have(name) & artifact("base", string baseName, cartago.ArtifactId id2) {
    string resName = baseName+"-"+name;
    cartago.makeArtifact(resName, "mams.passive.PassiveListArtifact",
        cartago.params([name, cls]), cartago.ArtifactId id);
    cartago.linkArtifacts(id, "out-1", id2);
    cartago.focus(id);
    cartago.operation(id, createRoute());
    +artifact(name, resName, id);
    +listResource(name, cls);
  }
}
agent Manager extends PassiveMAMSAgent {
  rule +!init() {
    MAMSAgent::!init();!created("base");
    !listResource("clients", "auction.Client");
    ...
  }
  rule $cartago.signal(string A,listItemCreated(string N,string T))
      : bidder_count(int cnt) {
    !monitorPassiveItem(N, T, A+"-"+N);
    string BN = "bidder_"+cnt+"_"+name;
    system.createAgent(BN, "Bidder"); +for_client(BN, A+"-"+N);
  }
}
```

Listing 4: Part of the mams.PassiveMAMSAgent program

and types. We view the creation of such resources as essential to support the implementation of concepts such as decentralised trust management [1] and social reputation [12].

- *Use of a Linked Data Model:* Linking of resources provides a way for external systems to discover and navigate complex APIs. Support for this has been realised through the use of the Hypertext Application Language (HAL) and through the adoption of agreed standards for representing the specific types of resource supported in this paper.
- *A Cleaner Approach to Resource Management:* The original MAMS model supported only one form of resource management, which was intimately linked to the agent program. The developer of the program was responsible for handling all HTTP requests. The model presented in this paper offers a more refined approach, where valid HTTP requests (those that are permitted for the given resource type) are vetted by the associated agent (invalid requests are automatically rejected). In this paper, we term this agent-in-the-loop approach *active resource management* (see Section 4.3).
- *A Passive Resource Management Model:* In addition to the active model, this paper introduces a *passive resource management* model that separates

agents from resource updates. Instead agents are passive observers that monitor their associated resources for change or who can modify the state of their resources directly (through internal operations that are equivalent to those supported by HTTP Requests).
– *Language Independence:* Finally, a last key advantage of the approach described in this paper is that it is agnostic to the agent programming language used. This has been achieve by focusing on an artifact-based model of virtual resources that is language independent.

Finally, [13] presents recent work on CArtAgO that exposes artifacts through a Web API. This contrasts with the work presented in this paper as it focuses first on simply exposing artifacts and secondly does so as a web API rather than as REST resources. As discussed in Section 4, the MAMS approach is quite different to the CArtAgO approach.

8 Conclusions

This paper presents a novel approach to the implementation of Multi-Agent MicroServices (MAMS), a model that sits at the intersection between Multi-Agent Systems and Microservices. The model embraces current industry best practice and technology stacks and proposes introduces the idea of virtual resources as a mechanism for facilitating the seamless integration of agents into microservices-based architectures. Through this, we gain access to a wealth of technologies and experience in how to deploy systems at scale while at the same time situating those agents in a larger web-enabled ecosystem.

Future work will seek to address a number of limitations of the model described here. This includes some improvements to the underlying architecture, but more significantly, the decoupling of resources and representations to allow multiple representations to be returned for a given resource. Ultimately, the aim is to support all the linked data formats described in [17] as well as any others that evolve over time. A specific target is the implementation of support for JSON-LD [28] representations which we intend to use in CONSUS[7], a research project that seeks, in part, to develop multi-agent decision-support tools for smart agriculture.

9 Acknowledgements

This research is funded under the SFI Strategic Partnerships Programme (16/SPP/3296) and is co-funded by Origin Enterprises Plc.

References

1. Aref, A.M., Tran, T.T.: A decentralized trustworthiness estimation model for open, multiagent systems (dtmas). Journal of Trust Management **2**(1), 3 (2015)

[7] http://www.consus.ie

2. Balalaie, A., Heydarnoori, A., Jamshidi, P.: Microservices architecture enables devops: Migration to a cloud-native architecture. Ieee Software **33**(3), 42–52 (2016)
3. Ciortea, A., Boissier, O., Ricci, A.: Engineering world-wide multi-agent systems with hypermedia. In: 6th International Workshop on Engineering Multi-Agent Systems (EMAS 2018) (2018)
4. 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. pp. 1659–1663. International Foundation for Autonomous Agents and Multiagent Systems (2019)
5. Collier, R.W., Russell, S., Lillis, D.: Reflecting on agent programming with agentspeak (l). In: International Conference on Principles and Practice of Multi-Agent Systems. pp. 351–366. Springer (2015)
6. Decker, S., Melnik, S., Van Harmelen, F., Fensel, D., Klein, M., Broekstra, J., Erdmann, M., Horrocks, I.: The semantic web: The roles of xml and rdf. IEEE Internet computing **4**(5), 63–73 (2000)
7. Dhaon, A., Collier, R.W.: Multiple inheritance in agentspeak (l)-style programming languages. In: Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control. pp. 109–120 (2014)
8. Dragoni, N., Lanese, I., Larsen, S.T., Mazzara, M., Mustafin, R., Safina, L.: Microservices: How to make your application scale. In: International Andrei Ershov Memorial Conference on Perspectives of System Informatics. pp. 95–104. Springer (2017)
9. Fielding, R.T.: REST: Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation (2000), http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm
10. Framework, Z.: Hypertext application language website. https://weierophinney.github.io/hal/hal/ (2019), accessed=29/10/2019
11. Griffiths, N., Chao, K.M.: Agent-based service-oriented computing. Springer (2010)
12. Hahn, C., Fley, B., Florian, M., Spresny, D., Fischer, K.: Social reputation: A mechanism for flexible self-regulation of multiagent systems. Journal of Artificial Societies and Social Simulation **10**(1) (2007)
13. International Foundation for Autonomous Agents and Multiagent Systems: Engineering Scalable Distributed Environments and Organizations for MAS (2019)
14. Kelly, M.: Json hypertext applicaion language specification. https://tools.ietf.org/html/draft-kelly-json-hal-08 (2016), accessed=29/10/2019
15. Kravari, K., Bassiliades, N.: Storm: A social agent-based trust model for the internet of things adopting microservice architecture. Simulation Modelling Practice and Theory **94**, 286–302 (2019)
16. Krivic, P., Skocir, P., Kusek, M., Jezic, G.: Microservices as agents in iot systems. In: KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications. pp. 22–31. Springer (2017)
17. Martins, J.A., Mazayev, A., Correia, N.: Hypermedia apis for the web of things. Ieee Access **5**, 20058–20067 (2017)
18. Mascardi, V., Weyns, D.: Engineering multi-agent systems anno 2025. In: International Workshop on Engineering Multi-Agent Systems. pp. 3–16. Springer (2018)
19. O'Brien, P.D., Nicol, R.C.: Fipa-towards a standard for software agents. BT Technology Journal **16**(3), 51–59 (1998)
20. O'Connor, R.V., Elger, P., Clarke, P.M.: Continuous software engineering-a microservices architecture perspective. Journal of Software: Evolution and Process **29**(11), e1866 (2017)

21. O'Neill, E., Lillis, D., O'Hare, G.M., Collier, R.W.: Explicit modelling of resources for multi-agent microservices using the cartago framework. In: Proceedings of the 18th International Joint Conference on Autonomous Agents and Multi-Agent Systems, Auckland, NZ, 2020. International Foundation for Autonomous Agents and MultiAgent Systems (IFAAMAS) (2020)
22. De la Prieta, F., Rodríguez-González, S., Chamoso, P., Corchado, J.M., Bajo, J.: Survey of agent-based cloud computing applications. Future Generation Computer Systems **100**, 223–236 (2019)
23. Rao, A.S.: Agentspeak (l): Bdi agents speak out in a logical computable language. In: European workshop on modelling autonomous agents in a multi-agent world. pp. 42–55. Springer (1996)
24. Ricci, A., Bordini, R.H., Hubner, J.F., Collier, R.: Agentspeak (er): An extension of agentspeak (l) improving encapsulation and reasoning about goals. In: The 17th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2018). International Foundation for Autonomous Agents and MultiAgent Systems (IFAAMAS) (2018)
25. Ricci, A., Viroli, M., Omicini, A.: Cartago: A framework for prototyping artifact-based environments in mas. In: International Workshop on Environments for Multi-Agent Systems. pp. 67–86. Springer (2006)
26. Roy, C.: Restful api design: Microserices. https://medium.com/@cknextmove/restful-api-design-microservices-f983e3ea3563, accessed: 2019-10-25
27. Savaglio, C., Ganzha, M., Paprzycki, M., Bădică, C., Ivanović, M., Fortino, G.: Agent-based internet of things: State-of-the-art and research challenges. Future Generation Computer Systems **102**, 1038–1053 (2020)
28. Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Lindström, N.: Json-ld 1.0. W3C recommendation **16**, 41 (2014)
29. Thönes, J.: Microservices. IEEE software **32**(1), 116–116 (2015)
30. Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: 2015 10th Computing Colombian Conference (10CCC). pp. 583–590. IEEE (2015)
31. W Collier, R., O'Neill, E., Lillis, D., O'Hare, G.: Mams: Multi-agent microservices. In: Companion Proceedings of The 2019 World Wide Web Conference. pp. 655–662. ACM (2019)
32. Xu, C., Zhu, H., Bayley, I., Lightfoot, D., Green, M., Marshall, P.: Caople: A programming language for microservices saas. In: 2016 IEEE Symposium on Service-Oriented System Engineering (SOSE). pp. 34–43. IEEE (2016)