Research Repository UCD

Title	MAMS: Multi-Agent MicroServices
Authors(s)	Collier, Rem, O'Neill, E. (Eoin), Lillis, David, O'Hare, G. M. P. (Greg M. P.)
Publication date	2019-05-17
Publication information	Collier, Rem, E. (Eoin) O’Neill, David Lillis, and G. M. P. (Greg M. P.) O’Hare. “MAMS: Multi-Agent MicroServices.” ACM, 2019.
Conference details	The 2019 World Wide Web Conference (WWW'19), San Francisco, United States of America, 13-17 May 2019
Publisher	ACM
Item record/more information	http://hdl.handle.net/10197/25818
Publisher's version (DOI)	10.1145/3308560.3316509

Downloaded 2024-05-27 09:30:16

The UCD community has made this article openly available. Please share how this access benefits you. Your story matters! (@ucd_oa)

© Some rights reserved. For more information

MAMS: Multi-Agent MicroServices*

Rem W. Collier
University College Dublin
Dublin, Ireland
rem.collier@ucd.ie

Eoin O'Neill
University College Dublin
Dublin, Ireland
eoin.o-neill.3@ucdconnect.ie

David Lillis
University College Dublin
Dublin, Ireland
david.lillis@ucd.ie

Gregory M.P. O'Hare
University College Dublin
Dublin, Ireland
gregory.ohare@ucd.ie

ABSTRACT

This paper explores the intersection between microservices and Multi-Agent Systems (MAS), introducing the notion of a new approach to building MAS known as Multi-Agent MicroServices (MAMS). Our approach is illustrated through a worked example of a Vickrey Auction implemented as a microservice.

CCS CONCEPTS

• **Information systems → RESTful web services**; • **Computing methodologies → Multi-agent systems**; • **Software and its engineering** → *Organizing principles for web applications.*

KEYWORDS

ACM proceedings, LaTeX, text tagging

ACM Reference Format:
Rem W. Collier, Eoin O'Neill, David Lillis, and Gregory M.P. O'Hare. 2019. MAMS: Multi-Agent MicroServices. In *Companion Proceedings of the 2019 World Wide Web Conference (WWW '19 Companion), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3308560.3316509

1 INTRODUCTION

As a research field, Multi-Agent Systems (MAS), has existed since the late 1980s. Since its inception, notions such as loose-coupling, distribution, reactivity and isolated (local) state have been core concepts [14, 45]. While the notion of an agent has been widely researched and a large range of tools and programming languages developed [1, 3, 38, 40], there has been little real adoption of these languages and tools within industry [8].

At the same time, industry has evolved through a range of enterprise paradigms and models, that have slowly shifted from monolithic centralised systems towards highly-decentralised systems that increasingly exhibit properties that were once seen as differentiators for MAS. Currently, it is the era of the microservice [20]; a model in which systems are built from small, loosely-coupled services that maintain their own independent state [13]. Collectively, these services are deployed within an ecosystem of tools and components that facilitate rapid and agile development techniques; are easy to extend; and support automated management of fault tolerance and scaling [46].

This paper argues that microservices represent a potential point of convergence between modern software engineering and MAS. It adopts the position that an agent can be viewed as a type of microservice that can be deployed seamlessly within any microservice ecosystem. From an external perspective, such a microservice would be indistinguishable from other standard microservices; with all interaction being directed through a uniform interface. This would allow the developers of such a service to leverage existing tools and components to deliver a MAS that is founded upon, and closely aligned with, industry accepted tools and platforms.

2 FROM MICROSERVICES TO MULTI-AGENT SYSTEMS

Microservices represent the state of the art for large-scale software development. While not a one-size-fits-all solution, they have clearly demonstrated their potential as a tool for building systems at scale. We believe that microservices have many similarities with MAS. In this section, we examine both approaches by exploring how the principles of microservices relate to MAS and vice-versa.

Microservices [20] are a realisation of the established Service-Oriented Architecture (SOA) design style [46]. Due to its popularity, many definitions of microservices pervade the web. In an effort to be complete, we have picked two. Firstly, we consider the definition of [13] who argues that the core principles of microservices are: Bounded Context, Size, and Independence. Secondly, we consider [46] who argues that one of the basic tenets of microservices is their adherence to the IDEAL (Isolated state, Distribution, Elasticity, Automated management and Loose coupling) design principles [15], a set of guiding principles for the design of all cloud-based software.

Table 1, presents a combined summary of these two definitions that reflects our attempt to understand how the principles of microservices relate to MAS. Due to their similarity, Independence and Loose Coupling have been combined.

Overall, the comparison demonstrates that there are many commonalities between these two approaches: isolated state, distribution, elasticity and loose coupling seem to be equivalent. From an agent perspective, isolated state and loose coupling arise from the view of agents as autonomous decision-makers [45] and the

*Produces the permission block, and copyright information

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.
WWW '19 Companion, May 13–17, 2019, San Francisco, CA, USA
© 2019 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC-BY 4.0 License.
ACM ISBN 978-1-4503-6675-5/19/05.
https://doi.org/10.1145/3308560.3316509

Table 1: Comparing Microservice principles to MAS

Principle	Microservices	MAS
Bounded Context	A microservice represents a single piece of business functionality.	An agent can play a single or multiple roles in a system.
Size	Microservices should be small enough to ensure maintainability and extensibility.	Size/complexity is not an issue in MAS research and often depends on the target domain.
Isolated State	Sharing of state information is minimised across services.	State is local and private to an agent. This is often viewed as essential for an agents autonomy [ref].
Distribution	Services are spread across multiple nodes.	Agents are logically distributed, but it is also expected that they will be spread over multiple nodes.
Elasticity	The application is designed to allow addition and removal of required resources at runtime.	The ability to add/remove agents at runtime is a central feature of MAS.
Automated Management	Management operations like failure handling and scaling are automated.	Management operations are not central to agents, but are sometimes considered.
Loose Coupling	Systems are decomposed into loosely coupled sets of highly-cohesive colocated services.	Agents are autonomous and loosely-coupled problem solvers.

distribution perspective comes from the view of agents as social entities [45] that can be considered "...a loosely-coupled network of problem solvers..." [14]. Elasticity, in terms of the ability to create or destroy agents at run-time has been a central concept in MAS research since its inception, for example [32].

The bounded context and size principles seem less consistent. In our opinion, this is a consequence of how each approach has emerged. Microservices are a practical response to the limitations of monolithic applications and the complexity of other service models. In contrast, MAS is dominated by research and encompasses a diverse range of approaches. This has resulted in less emphasis being placed on such issues. Automated management is an area that has seen some attention in MAS research, often under the guise of Autonomic Computing [24]. However, such techniques are rarely considered key in MAS research.

Another key tenet listed by [46] is polyglot programming - the argument that developers can and should use appropriate programming paradigms (e.g. Java, Python, Matlab) for each microservice.

Table 2: Comparing MAS principles to Microservices

Principle	Microservices	MAS
Autonomy	Microservices operate without the direct intervention of humans and have some kind of control over their internal state (and actions?).	Agents operate without the direct intervention of humans and have some kind of control over their actions and internal state.
Social Ability	Interaction between microservices is typically achieved using messages based on RESTful APIs and HTTP.	Agents interact with other agents using some kind of Agent Communication Language.
Reactivity	Microservices respond to incoming HTTP requests in a timely fashion.	Agents perceive their environment and respond in a timely fashion to changes that occur in it.
Proactivity	Microservices do not take the initiative.	Agents don't just respond - they take the initiative.

To offer an opposing analysis, we consider how well microservices conform to agnet concepts. For this, we choose the classic weak notion of agency espoused by Wooldridge and Jennings [45] which argues that an agent is any entity that demonstrates autonomy, social ability, reactivity and proactivity. Table 2 compares how these features apply to both microservices and agents, with the MAS definitions taken directly from their paper.

Microservices seem to meet the autonomy, social ability and reactivity principles quite well. The differences revolve mainly around the fact that agents are situated entities, while microservices are not. From an MAS perspective, reactivity is closely related to the agents' environment. From a Microservices perspective, reactivity is simply the ability to respond effectively to incoming HTTP Requests. While they are clearly not the same level of complexity, if we view a microservices environment to be the set of messages it receives, then clearly it is reactive.

The view of microservices as entities that react just to messages is one that is being challenged in the context of Web of Things style applications [7] where things are represented as microservices that expose state and enable remote action [23]. While this reduces the conceptual gap for autonomy, it is not reflected in reactivity. This is because microservices principles are concerned primarily with defining external rather than internal attributes. As such, microservices do not consider how their internal state is realised and maintained, just that the state exists, is represented as a set of resources, and can be accessed/modified though a uniform (REST) interface. It is this same consideration that affects the proactivity principle - microservices principles do not cover proactivity because it is an internal quality. Finally, while social ability might seem compatible, it is less so when reflecting on the core objective of each approach: microservices aim to directly expose

(data) resources in a fine grained way, while agents seek to expose knowledge (state) and capability (action).

3 MULTI-AGENT MICROSERVICES (MAMS)

This section introduces *Multi-Agent MicroServices (MAMS)*, a class of system that is comprised of Agent-Oriented MicroServices (AOMS), and Plain-Old MicroServices (POMS). AOMS are microservices that are built using MAS technologies which are exposed through a well-defined interface modelled as a set of REpresentational State Transfer (REST) [16] [17] endpoints Such a system is an essential step in achieving our longer-term goal of realising Hypermedia MAS [7] - agent-based systems that are able to discover, consume and integrate hypermedia services which act as an enabler for Semantic Web and Linked Data systems [36, 41]

3.1 Agents and Microservices

In this section, we consider the relationship between agents and microservices. To do this, we first reflect on the relationship between resources and microservices that is promoted through REST.

Conceptually, REST promotes a view in which all resources are isolated as distinct services. In practice, resources are often co-located to minimize the impact of issues such as `chattiness` [39], which occurs where resources have been overly decomposed resulting in an abundance of network calls. This relates to the issue of Bounded Context discussed in section 2. The result of this is that a single microservice may often play host to multiple resources.

How does this relate to agents? Conceptually, agents are not just resources; they are complex decision-making entities that reason about their actions. On the other hand, agents rarely work in isolation, it is a common model that groups of agent work closely together through some form of collaborative process that is often realised through message-passing (e.g. auctions). As such, it is clear that such agents may suffer from similar issues, such as, chattiness. As with resources, it may make sense to co-locate such agents within a single microservice.

What about the relationship between agents and resources? Although there is a clear difference between agents and resources, we adopt a view that an agent is something that can both implement resources and be exposed as resources. For example, the inbox of an agent can be exposed as a resource, with other agents being able to interact with it by sending a POST request to an inbox URL, creating a new message. Alternatively, an agent could expose aspects of its internal state as resources. For example, a bidding agent that participates in an auction could expose its bidding strategy as a resource through a well-defined URL. External systems would then be able to update the strategy by sending a PUT request to that URL, updating the bidding strategy of the agent. In this sense, an AOMS is a community of autonomous agents that expose a related set of resources.

In terms of exposing resources, we adopt the view that each resource is associated with exactly one agent. That said, not every agent needs to be associated with a resource. This offers an interesting dichotomy: an AOMS consists of a set of agents that are mapped to resources and a set of agents that are not. In essence, agents that expose resources form part of the interface of the microservice and must be externally accessible. All other agents are part of the implementation detail and so do not require an external interface. This view offers many parallels with human organisations. For example, in a company, sales and customer support staff are engaged to interact with the public while many other staff are engaged to implement internal functions necessary to run the company. From a software perspective, it also correlates nicely with notion of visibility: externally facing agents can be classed as public agents and all others as private agents.

Building on these notions, we introduce the idea of a class of AOMS that represents reusable components of larger microservices architectures. These AOMS would provide standarised implementations of common auctions [21, 22] and organisational structures [11]. The internal mechanics of these implementations would use agents, but the external interface would be realised as a set of REST resources, allowing both AOMS and POMS to use them. We term this approach **Organisation as a Service (OaaS)**.

3.2 Social Ability and MAMS

Social ability is a key requirement of MAS - it represents the impetus to move away from individual problem solvers towards networks of collaborative/cooperative problem solvers. Traditionally, interaction has been modelled through the use of Agent Communication Languages (ACLs) like FIPA-ACL [19]. More recently, the perceived failures of these ACLs has seen the emergence of new breeds of ACL, such as the Blindingly Simple Protocol Language (BSPL) [42]. Alternatively, other researchers have argued for other modes of interaction based on concepts like shared Artifacts [35], which consider support for interaction to be the responsibility of the environment. This style of model has been discussed in detail in the context of adopting the environment as a first class entity in MAS development [44]. In practice, it has become clear that the adoption of a single approach is often insufficient and many MAS now promote a hybrid communication strategy that combines both approaches. This hybrid view fits well with a microservices approach. As such, we consider a number of complementary modes:

- *Interaction through RESTful APIs.* One potential model of multi-agent interaction discussed in Section 3.1 is to expose the knowledge (state) and capabilities (actions) of an agent directly as resources. Such a model is potentially appealing because it would allow any other component to interact seamlessly and transparently with an AOMS through the associated RESTful API. In such a model, an agent could manage *virtual resources* that are internal abstractions of the agent's internal states or the state of the environment it inhabits. This state is encoded within the agent's beliefs, which are maintained through perception. Changes to the virtual resource through RESTful calls could lead to changes in the agent's beliefs and ultimately drive its actions.
- *Interaction through shared artifacts.* POMS represent resources that are exposed through well-defined interfaces. This is somewhat similar to the notion of an artifact as described in [35]. State updates can be easily achieved through REST operations. It is precisely this model that we envisaged in Section 3.1 where we discussed the idea of a bidding strategy resource being used to allow external systems to update their bidding strategy for an auction.

- *Interaction through high-level communication.* AOMS can represent single or multi-agent systems. Within an AOMS, each externally accessible (public) agent can be uniquely identified by a Uniform Resource Identifier (URI). While we have previously focused on the idea that agents could expose "virtual resources", it is also possible to view the inbox of an agent as a concrete resource, and message passing as being equivalent to a POST operation applied to that resource. Such a model could apply to both locally (the view adopted in this paper) and remotely hosted inboxes [5]. To be clear, this approach is different to the pure RESTful API based communication discussed earlier in that it is more constrained. Specifically, high-level communication refers to communication based on human models of interation [6] (e.g. FIPA-ACL [19]).
- *Interaction through conversations.* As an interesting next step to exposing an agent's inbox as a resource, it is possible to introduce the notion of an agent whose protocols and conversations are exposed as resources. This could be a particularly appealing model as it would allow other agents to understand what protocols an agent knows (possibly including semantic descriptions of those protocols). Additionally, links representing valid responses could be associated with the messages to direct the flow of the conversation. Such a model could build on internal conversation managers like the Agent Conversation Reasoning Engine [30, 31]. For example, if agent A initiates a conversation based on the FIPA Request Protocol with agent B, then A would generate a conversation resource (and associated URI) that encapsulates the conversation. This would make the conversation persistent, referable and lend itself concepts such as accountability and normative behaviour [43].

3.3 Leveraging an Industry Strength Tool Ecology

Microservices has established itself as the leading approach to developing large scale systems in industry [12]. It is recognised as a key tool in managing complexity, maintaining agility and improving the effectiveness of the development team [34]. Key to this success are a suite of tools and components that help to ease the development and deployment process:

- *Containers:* Containerisation sits at the forefront of modern distributed computing, representing a lightweight class of virtualisation platforms. Containers provide a lightweight portable run-time while allowing for development, testing and deployment and the ability to communicate between containers [37]. Key to their success is the focus on single applications/services rather than full Operating Systems coupled with libraries of pre-configured images of services that can be easily downloaded, deployed and tweaked to meet current needs. The dominant implementation of container technology is Docker[1].
- *Container Orchestration:* Effective deployment and management of containerized systems is a key issue. This has led to the emergence of orchestration tools that automate the management and monitoring of large clusters of containers. The dominant player in this space is Kubernetes[2].
- *Infrastructure Services:* The effective deployment of microservices requires more than container orchestration. A range of additional infrastructure services are needed to realise truly reliable deployments. For example, [25] argues that a container orchestrator should facilitate scalability, load balancing, service deployment and discovery, and possibly service migration. [10] adds concepts such as message security, service proxies and data storage infrastructures. For example, when Netflix re-engineered their system to adopt a microservices approach, they were forced to develop a range of technologies that previously were not available [29], including: a load balancer called "Ribbon", and a discovery service called "Eureka".
- *Design Patterns:* Increased adoption of microservices had led to emerging awareness of what works and what does not. Architectural design patterns have been proposed encoding this best practice [10], with the most prominent patterns being: API Gateway, Publish/Subscribe, Circuit Breaker, Proxy and Load Balancer. Many other patterns are discussed online[3]. It is interesting to note that the API Gateway, Pub/Sub and Proxy patterns correspond to KQML facilitator patterns proposed in the early 1990s [18].

4 ILLUSTRATING MAMS

To provide some context for our approach, we have built a prototype based on ASTRA [8, 9], a variation of the AgentSpeak(L) programming language [4, 38]. ASTRA was used because it offers a closer integration with Java than many other AOP languages [8] as it exposes object references to the logical layer of the agent. This allows the direct representation of relations between those objects. It also provides a custom event model (part of its module mechanism) that enables developers to introduce new event types into the language. These features have proven particularly useful in the development of this prototype.

In the creation of this prototype, 4 main aims have been achieved: the provision of a mechanism for implementing a REST interface to virtual resources managed by an agent; the provision of a mechanism that enables agents to interact directly with REST resources; the exposing of the agents inbox as a REST resource; and the illustration of the MAMS concept.

4.1 Instantiating the MAMS model

Before launching into the technical (and ASTRA) specific details of our prototype implementation, we start with some more general refinements that could be applied to any prototype implementation. Specifically, we make the following assumptions:

- As is normal practice, it is assumed that each microservice will run in a separate Java Virtual Machine hosting a single HTTP server. In this way, each microservice instance can be referenced by the hosts name and the port number through which the HTTP server is exposed.

[1]http://docker.com

[2]https://kubernetes.io/

[3]http://microservices.io

- Each public agent will register with and be exposed by this web server. The agent will be treated as a resource and associated with a globally unique URL of the form:
`http://<host:port>/<agent-name>`

In our view the agent should not be directly addressable as a REST resource. By this, we mean that you cannot perform HTTP operations on the agent directly. Instead, we argue that an agent is a container for resources and its URL should act as a base for the creation of resource-specific URLs. In our implementation, these will be either the agent's inbox or other virtual resources associated with the agent:

- Each public agent inbox will be exposed through a URI:
`http://<host:port>/<agent-name>/inbox`
- Virtual resources associated with an agent will be exposed using URLs based on that agents base URL:
`http://<host:port>/<agent-name>/<resource-path>`

A possible future step would be to consider how additional aspects of an agent could be exposed. For example a `/beliefs` URI could return a representation of the agents beliefs. More interestingly, this URI could return the **public** beliefs of the agent allowing an agent to share some of its beliefs with other agents while hiding others. The same could be applied to goals or intentions enabling agents to reason about each others beliefs and activities.

It is worth noting that the URL of an agent is, by definition, globally unique. In our view, this is a simple and elegant solution to the issue of ensuring uniqueness of agent names. The way it is used to expose the agents inbox also contrasts with the approach advocated in the FIPA standards [19], where messages are posted to a URL that represents the agent communication channel (ACC) for the platform which, in turn, is responsible for delivering the message to the agent. This has a downside, in that the agent is potentially less mobile (because its identity is bound to a specific URL), however these problems are well understood on the web, and existing mechanisms such as URL redirection could be used to alleviate such shortcomings.

4.2 Extending ASTRA for AOMS

To illustrate some aspects of our proposed approach, we have adapted the ASTRA programming language to support the creation of virtual resources and FIPA-ACL based communication via RESTful inboxes. This was achieved through the use of ASTRA modules - Java classes that allow the developer to create custom actions, terms, formulae, sensors, and events [?] through annotated methods. Specifically, we developed a `Http` module that provides an interface to a Java Web Server implemented using Netty[4]; a non-blocking IO library.

The Web Server is implemented using the singleton pattern enforcing the 1 server per Java Virtual Machine (JVM) policy. The `Http` module includes two actions to allow creation of the singleton on either a default port (9000) or a user-specified port. An additional action is provided that enables agents to register with the Web Server. This exposes the agent as was defined in Section 4.1.

As was described in the previous section, valid REST resource URLs include the inbox URL or the URL of any virtual resource associated with the public agents. Incoming HTTP requests are handled in one of 2 ways:

- For requests targeted at an agent's inbox, a custom piece of code is executed that transforms the body of the incoming request into a FIPA-ACL message that is passed directly to the relevant agent. This only occurs if the HTTP request is a POST request and a 200 OK response is returned. All other HTTP verbs result in a 403 Forbidden response. To send messages, we implement a custom *ASTRA Message Service*. This is a component of ASTRA that is used to implement bespoke messaging infrastructures. Specifically, this component is used to implement a mechanism that transforms a FIPA-ACL message into a JSON payload that is send as a HTTP POST to a specified URL (another agent's inbox).
- For all other requests, a custom `Http` event is generated. Event types currently exist for only four HTTP verbs: POST, GET, PUT, DELETE. However, it is trivial to extend this to cover all relevant HTTP verbs. The custom `Http` events contain 3 parameters. Two are references to Java objects that represent the incoming request and a context that can be used to generate the HTTP response. The third parameter is a (logical) list of strings that represent the remaining segments of URL associated with the HTTP request that is being handled. These are the segments that come after the part of the URL that equates to the agent's name. We illustrate how custom events are used in Section 3.3.

4.3 Vickrey Auction as a Service

To illustrate our approach, we implement a Vickrey Auction as a Service that can be used by other microservices. A high-level view of the service can be seen in Figure 1. The implementation consists of three types of agent: a Manager agent, Auctioneer agents, and Bidder agents. A single manager is created (with name "manager"). It is responsible for managing two virtual resources:

- `/manager/items`: a resource that represents all items being sold using the service. External services wishing to use the auction service submit POST requests to this URI containing details of items to be sold. Currently, the item model includes a name, a quantity and a reserve price (that must be reached for the item to be sold). The manager agent assigns a unique id to each incoming request. The `/items/{id}` URL can be used to access the item once it has been created.
- `/manager/clients`: a resource that represents all services that are interested in participating in auctions run through the service. Upon registration, a bidder is created for each client microservice. This bidder is the advocate that acts on behalf of the client service. Interaction between the client and the bidder is mediated via a `/wanted` resource through which the client can indicate what items it is interested in and what its bidding strategy should be for each item (e.g. how many items, and how much to pay per item).

To more clearly illustrate how the service works, we will walk through an example sequence of interactions. First we explore how a client microservice (named "MS1") would interact with the auction service. The key steps are illustrated through the arcs labelled 1-5 in Figure 1:

[4]http://netty.io

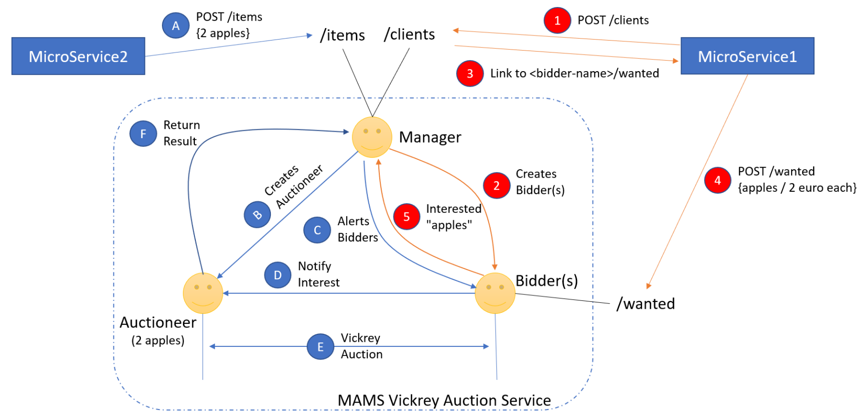

Figure 1: Illustration of the Vickrey Auction as a Service

- **1**: MS1 submits a POST request to `/manager/clients` registering its interest in using the service.
- **2**: The manager verifies MS1, assigns it a unique id and creates a bidder agent matched to that id.
- **3**: The manager responds to the initial POST with a `200 OK` response and the URL of the `<bidder-name>/wanted` resource that has been created specifically for MS1.
- **4**: MS1 submits a post request to the `/wanted/` resource indicating that it wants to purchase 2 apples and is willing to pay 2 euro per apple.
- **5**: MS1's bidder agent registers its interest in auctions of apples with the manager agent.

At this point, an agent has been created to act on behalf of MS1 and it has been assigned the task of buying 2 apples. No further action will take place until another microservice (MS2) decides to sell some apples. To understand the process undertaken by MS2, we will explain the key steps labeled A-F in figure 1:

- **A**: MS2 submits a POST request to `/manager/items` providing details of a type of item it wished to sell (in this case 2 apples). The manager responds immediately with a 200 OK and a link to the item created (through which MS2 can monitor the sales process).
- **B**: At some later point in time, the manager starts the auction process for the item by creating an Auctioneer agent. Tthe delay could be due to a limit on the number of concurrent auctions permitted on the platform, or simply because no other services are currently interested in the item being sold.
- **C**: Once the Auctioneer agent is created, the manager informs all interested bidders that an auction is about to start.
- **D**: Bidder agents decide whether or not to participate. Those that wish to inform the Auctioneer of their interest.
- **E**: The Auctioneer executes a standard Vickrey Auction between all interested bidder agents.
- **F** Details of the winning bid are returned to the manager who updates the item record to reflect the outcome and the winning Microservice is notified of their success.

Figure 2 contains a snippet of ASTRA code that relates to steps B and C above. It implements a simplified solution: the manager agent does not consider whether there are interested agents, but instead creates an auctioneer agent and tells all interested agents about it (if there are no interested agents, then it tells nobody). It executes auctions sequentially. The auctioneer waits 5 seconds to allow interested agents to register their intention to bid, and then starts the auction, in which it waits for a further 10 seconds before assessing all received bids. The highest bid (if one exists) is accepted and the items are sold to the associated microservice.

To deploy our MAMS implementation, it was necessary to make a number of significant changes to the way that ASTRA is compiled and deployed. Traditionally, ASTRA has been supported through a custom Eclipse plugin that integrates with the standard Eclipse build process. This has proven an effective way of allowing developers to write and execute ASTRA programs. However, this build process is not well suited to microservices-based approaches which tend to follow the Continuous Integration/Continuous Delivery (CI/CD) model. To remedy this, we have restructured ASTRA into a set of Maven[5] artifacts. Further, we have re-engineered the ASTRA Compiler to work as part of a Maven Plugin that can be easily integrated into its build lifecycle. Making these changes has allowed us to follow industry best practice to automate the build process and support the automatic creation and deployment of MAMS services. Specifically, we have created a Docker image of our auction system that can be deployed as required.

While the above service is quite simple, we believe that it demonstrates a number of interesting properties:

- We have built and deployed a MAMS that can be used seamlessly by other microservices. This can be easily extended to include external agents through the use of FIPA-ACL and public inboxes.
- By using the MAMS approach, we have implemented a Vickrey Auction that is self-contained and reusable. Other auctions or organisational mechanisms can be implemented in the same way to provide a library of reusable solutions.
- MAMS exposes only the agents/resources that need to interact with external systems. As a result, the Auctioneer, which implements the auction protocol (and decides the winner)

[5]http://maven.apache.org

```
agent Manager {
    ...
    rule $http.post(ChannelHandlerContext ctx,
        FullHttpRequest req, ["items"],
        string bdy) {
      Item item = il.itemFromJson(bdy);
      il.storeItem(item, string id);

      !!auctionItem(id, il.getItemName(item));

      ResponseObject obj = http.createResponse();
      http.setStatus(obj, 200);
      http.setLocation(obj,
          http.myAddress()+"/items/"+id);
      http.sendResponse(ctx, req, obj);
    }

    synchronized rule +!auctionItem(string id,
        string item) {
      !auctioneer("auctioneer"+id, item);
      foreach (interest(string name, item)) {
        send(inform, name,
            available(item, "auctioneer"+id));
      }
    }
    ...
}
```

Figure 2: ASTRA code implementing sequential auctions

is an internal (private) agent. We believe that this offers a more secure solution because it is not possible for agents or microservices (other than the manager and bidders) to interact directly with the Auctioneer.

5 RELATED WORK

Agents and microservices are garnering increased interest within the research community. For example, [26] present a case study in which agents are combined with microservices to implement rule-based eCommerce applications for IoT. The agent aspect of the approach is implemented using EMERALD [27]: a knowledge-based framework built on top of JADE [1].

[28] proposes a collaborative microservices-based model for IoT systems. In their paper, the authors argue, as we do, that microservices can be viewed as being similar to multi-agent systems. However, their approach is more of a design perspective than an implementation strategy and they do not propose the use an concrete agent technologies as we have done in this paper.

[43] offers a view of decentralized multi-agent systems for IoT where agents act as the head of a distributed body consisting of multiple hetereogenous IoT devices. In this paper, sharing of resources (devices) is realised through a normative interaction model that applies positive and negative sanctions in response to interaction. While superficially similar, our approach is really quite different - it does not require agents to act as the head of a service, instead, services and agents are seen as equal partners that are allowed to inter-operate freely as needed by the application. This, when combined with the layered model preferred for REST promotes a view of application that are comprised of a combination of agent based and non-agent based services.

[2] argues for the creation of stateful cloud-native microservices using Akka and Kubernetes. The report clearly highlights the importance of container and container orchestration technologies to modern distributed systems development. This view is one presented by a leading proponent of the Reactive Services/Akka movement. Their clear message is that for technology to be used it industry, it must fit the industry models and views. This paper is our attempt to present that view for MAS technologies.

6 CONCLUSIONS

In this paper, we have argued that adopting a Multi-Agent MicroServices (MAMS) approach offers the potential to deliver truly decentralized multi-agent applications. The specific approach we have used to demonstrate MAMS, described in section 4, is not an attempt to offer a technological solution but instead a vehicle for understanding what a MAMS system may look like and how it may operate. For this reason, we have not attempted to formalization MAMS but instead have followed a more discursive approach.

We conclude by reflecting on the implications of our approach. From a MAS perspective, we believe that some of the main benefits / implications include:

- The embracing of Web models to provide a simple decentralized global unique naming system for agents can act as a basis for driving new decentralized approaches to building MAS. For example, exposing aspects of an agents state and permitting other agents to reason about that state or providing discoverable semantic descriptions that can be used to allow agents to reason about how to interact with one another.
- The decomposition of agent systems into self-contained subsystems that can be independently tested has the potential to improve reliability and promotes the creation of reusable libraries of agents [33].
- The alignment of MAS deployment with industry best practice allows us to leverage the vibrant ecosystem of industry standard tools and components available to microservices based systems and where appropriate adapt them for use with MAS.

From a microservices perspective, we believe that the main benefit is that MAMS enables the seamless integration of agent technology into microservices-based systems. For example, the Organization as a Service (OaaS) model proposed in section 3.1 argues for the creation of shrink-wrapped implementations of various organizational structures that can be used by traditional microservices as mechanisms for mediating access to one another. This can range from implementations of auctions such as the one demonstrated in 4 to fully fledged network or market patterns [11].

ACKNOWLEDGMENTS

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

REFERENCES

[1] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. 1999. JADE âĂŞ A FIPA-compliant Agent Framework. *Fourth International Conference on Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 1999)* (1999), 97–108. https://doi.org/10.1145/375735.376120

[2] Jonas Boner. 2019. How To Build Stateful , Cloud-Native Services With Akka And Kubernetes Tips for running your stateful services as if they are stateless by Jonas Bonér. (2019). https://www.lightbend.com/white-papers-and-reports

[3] Rafael H Bordini, Lars Braubach, Jorge J Gomez-sanz, Gregory O Hare, Alexander Pokahr, and Alessandro Ricci. 2006. A Survey of Programming Languages and Platforms for Multi-Agent Systems. 30 (2006), 33–44.

[4] Rafael H Bordini, Jomi F Hübner, and Renata Vieira. 2005. {Jason} and the Golden Fleece of Agent-Oriented Programming. *Multi-Agent Programming — Languages, Platforms and Applications* 15 (2005), 3–37. https://doi.org/10.1007/0-387-26350-0_1

[5] Jean Paul Calbimonte, Davide Calvaresi, and Michael Schumacher. 2018. Multi-agent interactions on the web through linked data notifications. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-030-01713-2_4

[6] B Chaib-Draa, A Ken, J Williams, C Hall, and R Kent. 1994. Distributed artificial intelligence: An overview. *Encyclopedia Of Computer Science And Technology* 31 (1994), 215–243.

[7] Andrei Ciortea, Olivier Boissier, and Alessandro Ricci. 2018. Engineering World-Wide Multi-Agent Systems with Hypermedia. In *6th International Workshop on Engineering Multi-Agent Systems (EMAS)*. http://www.fipa.org/repository/standardspecs.html,

[8] R.W. Collier, S. Russell, and D. Lillis. 2015. *Reflecting on agent programming with agentspeak(L)*. Vol. 9387. https://doi.org/10.1007/978-3-319-25524-8_22

[9] A. Dhaon and R. Collier. 2014. Multiple inheritance in AgentSpeak(L)-style programming languages. In *AGERE! 2014 - Proceedings of the 2014 ACM SIGPLAN Workshop on Programming Based on Actors, Agents, and Decentralized Control, Part of SPLASH 2014*. https://doi.org/10.1145/2687357.2687362

[10] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. 2017. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017* (2017), 21–30. https://doi.org/10.1109/ICSA.2017.24

[11] Virginia Dignum and Frank Dignum. 2001. Modelling Agent Societies: Co-ordination Frameworks and Institutions. *Progress in Artificial Intelligence* 2258 (2001), 7–21. https://doi.org/10.1007/3-540-45329-6_21 arXiv:arXiv:hep-th/0112055v2

[12] Nicola Dragoni, Saverio Giallorenzo, Alberto Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, Larisa Safina, Nicola Dragoni, Saverio Giallorenzo, Alberto Lafuente, Manuel Mazzara, Fabrizio Montesi, Manuel Mazzara, and Bertrand Meyer Present. 2017. Microservices : yesterday , today , and tomorrow. *Present and Ulterior Software Engineering* (2017).

[13] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina. 2018. Microservices: How to make your application scale. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-319-74313-4_8 arXiv:1702.07149

[14] Lesser V.R. Durfee, E.H. 1988. Incremental Planning to ControI Time-Constrained Blackboard-Based Problem Solver. *IEEE TRANSACTIONS ON AEROSPACE AND ELECTRONIC SYSTEMS* 24, 5 (1988).

[15] Christoph Fehling. 2015. *Cloud Computing Patterns Identification, Design, and Application*. Technical Report.

[16] Roy T Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation. University of California, Irvine.

[17] Roy T Fielding and Richard N Taylor. 2002. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology* 2, 2 (2002).

[18] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. 1994. KQML as an agent communication language. *Proceedings of the third international conference on Information and knowledge management - CIKM '94* (1994), 456–463. https://doi.org/10.1145/191246.191322

[19] FIPA. 2000. FIPA Standards. http://www.fipa.org

[20] Martin Fowler. 2014. MicroServices: A definition of this new architectural term. https://martinfowler.com/articles/microservices.html

[21] R Guttman and P Maes. 1998. Cooperative Information Agents II, Learning, Mobility and Electronic Commerce for Information Discovery on the Internet, Second International Workshop, CIA' 98, Paris, France, July 4-7, 1998, Proceedings. *Cia* 1435 (1998), 135–147. https://doi.org/10.1007/BFb0053669

[22] Fu Shiung Hsieh. 2006. Analysis of contract net in multi-agent systems. *Automatica* 42, 5 (2006), 733–740. https://doi.org/10.1016/j.automatica.2005.12.002

[23] Muhammad Aslam Jarwar, Sajjad Ali, Muhammad Golam Kibria, Sunil Kumar, and Ilyoung Chong. 2017. Exploiting interoperable microservices in web objects enabled Internet of Things. In *International Conference on Ubiquitous and Future Networks, ICUFN*. https://doi.org/10.1109/ICUFN.2017.7993746

[24] Jeffrey O Kephart and David M Chess. 2003. The vision of autonomic computing. *Computer* 1 (2003), 41–50.

[25] Nane Kratzke. 2014. A lightweight virtualization cluster reference architecture derived from open source paas platforms. *Open Journal of Mobile Computing and Cloud Computing* 1, 2 (2014), 17–30.

[26] Kalliopi Kravari and Nick Bassiliades. 2018. A Rule-Based eCommerce Methodology for the IoT Using Trustworthy Intelligent Agents and Microservices. https://doi.org/10.1007/978-3-319-99906-7_22

[27] Kalliopi Kravari, Efstratios Kontopoulos, and Nick Bassiliades. 2010. EMERALD: A multi-agent system for knowledge-based reasoning interoperability in the semantic web. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6040 LNAI (2010), 173–182. https://doi.org/10.1007/978-3-642-12842-4_21

[28] Petar Krivic, Pavle Skocir, Mario Kusek, and Gordan Jezic. 2018. Microservices as agents in IoT systems. In *Smart Innovation, Systems and Technologies*. https://doi.org/10.1007/978-3-319-59394-4_3 arXiv:arXiv:1607.08131v1

[29] Andrew Leung, Andrew Spyker, and Tim Bozarth. 2017. Titus: Introducing Containers to the Netflix Cloud. *Queue* 15, 5 (2017), 30.

[30] David Lillis. 2012. *Internalising Interaction Protocols as First-Class Programming Elements in Multi Agent Systems*. Ph.D. Dissertation. University College Dublin. https://doi.org/10.13140/RG.2.1.1573.7040/2

[31] David Lillis, Rem W. Collier, and Howell R. Jordan. 2013. Evaluation of a Conversation Management Toolkit for Multi Agent Programming. In *Programming Multi-Agent Systems - 10th International Workshop, ProMAS 2012, Valencia, Spain, June 5, 2012, Revised Selected Papers*, Mehdi Dastani, Jomi F. Hübner, and Brian Logan (Eds.). Vol. 7837. Springer Verlag Heidelberg, 90–107. https://doi.org/10.1007/978-3-642-38700-5_6

[32] David Lillis, Rem W. Collier, Fergus Toolan, and John Dunnion. 2007. Evaluating Communication Strategies in a Multi Agent Information Retrieval System. In *Proceedings of the 5th European Workshop on Multi-Agent Systems (EUMAS '07)*. Hammamet, Tunisia.

[33] Michael Luck, Peter McBurney, and Chris Preist. 2003. *Agent technology: enabling next generation computing (a roadmap for agent based computing)*. AgentLink.

[34] Ekaterina Novoseltseva. 2019. Benefits of Microservices Architecture Implementation. https://dzone.com/articles/benefits-amp-examples-of-microservices-architectur

[35] Andrea Omicini, Alessandro Ricci, Mirko Viroli, Cristiano Castelfranchi, and Luca Tummolini. 2004. Coordination Artifacts: Environment-based Coordination for Intelligent Agents. *International Joint Conference on Autonomous Agents and Multiagent Systems 2004)* (2004), 286–293. https://doi.org/10.1109/AAMAS.2004.10070

[36] Kevin R Page, David C De Roure, and Kirk Martinez. 2011. REST and Linked Data: a match made for domain driven development?. In *Proceedings of the Second International Workshop on RESTful Design*.

[37] Claus Pahl and Brian Lee. 2015. Containers and clusters for edge cloud architectures–A technology review. In *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. IEEE, 379–386.

[38] A. Ricci, R.H. Bordini, J.F. Hubner, and R. W. Collier. 2018. AgentSpeak (ER): An Extension of AgentSpeak (L) improving Encapsulation and Reasoning about Goals. *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems* (2018).

[39] Mark Richards. 2015. *Microservices vs. service-oriented architecture*. O'Reilly Media.

[40] S. Russell, H. Jordan, G.M.P. O'Hare, and R.W. Collier. 2011. *Agent factory: A framework for prototyping logic-based AOP languages*. Vol. 6973 LNAI. https://doi.org/10.1007/978-3-642-24603-6_13

[41] Ivan Salvadori, Alexis Huf, Ronaldo dos Santos Mello, and Frank Siqueira. 2016. Publishing linked data through semantic microservices composition. *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services - iiWAS '16* November 2017 (2016), 443–452. https://doi.org/10.1145/3011141.3011155

[42] Munindar P. Singh. 2014. Bliss: Specifying declarative service protocols. In *Proceedings - 2014 IEEE International Conference on Services Computing, SCC 2014*. https://doi.org/10.1109/SCC.2014.39

[43] Munindar P Singh and Amit K Chopra. 2017. The internet of things and multiagent systems: Decentralized intelligence in distributed computing. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1738–1747.

[44] Danny Weyns, Andrea Omicini, and James Odell. 2007. Environment as a first class abstraction in multiagent systems. *Autonomous Agents and Multi-Agent Systems* 14, 1 (2007), 5–30. https://doi.org/10.1007/s10458-006-0012-0

[45] Michael Wooldridge and Jennings Nicholas R. 1995. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review* 10, 2 (1995).

[46] Olaf Zimmermann. 2017. Microservices tenets: Agile approach to service development and deployment. *Computer Science - Research and Development* (2017). https://doi.org/10.1007/s00450-016-0337-0