FLASH-MAS: towards a state-of-the-art, modular agent deployment environment

Andrei Olaru[0000−0002−2718−9195]

Department of Computer Science and Engineering, National University of Science and Technology POLITEHNICA Bucharest, 313 Splaiul Independentei, 060042 Bucharest, Romania andrei.olaru@upb.ro

Abstract. Agent development and deployment frameworks have seen little development recently, and by far the most popular framework is Jade, with SPADE coming strong for Python deployments.
Flash-mas – the Flexible, Lightweight Agent Shell – is a framework for the deployment of multi-agent systems comprising a variety of entities besides agents, with a great level of modularity. In Flash-mas, any part of the system – agents, but also nodes, communication infrastructures, services, and agent components – can have an arbitrary implementation and is loaded dynamically when a scenario is started. Moreover, any other type of entity is automatically, dynamically integrated, and can easily interact with other entities on the same node or on other nodes.
We present in this demonstration our current progress in the development of Flash-mas, how new types of entities can be seamlessly integrated in a deployment, how machine learning models written in Python can integrate with Java agents, how remote visualization is possible via a web application, and how monitoring the progress of a scenario is possible thanks to log summarizers and finite state machines.

Keywords: MAS framework · visualization · flexibility · modularity

1 Introduction

The current MAS framework landscape is mostly divided into three areas: agent-based model simulation (ABMS), led by Repast [4]; FIPA-compliant [5] deployment that can be network-distributed, led by Jade in Java [3] and SPADE in Python [8]; and, more recently, agentic AI initiatives currently coagulating around protocols such as MCP and A2A [1,2].

We introduced Flash-mas[1] [6] several years ago as a MAS deployment and execution framework that is modular and flexible. The target of Flash-mas is to be a general-purpose MAS deployment framework that can be used in various scenarios: as an alternative to (or integrated with) Jade; as a competent ABMS platform; and as an infrastructure for running agentic AI applications. Our approach follows a few basic principles:

[1] The Flash-mas repository is located at https://github.com/andreiolaru-ro/FLASH-MAS.

```
-package testing -loader agent:composite
-node node1
  -pylon webSocket:pylon1 serverPort:8886
    -agent composite:agA -shard messaging -shard PingTest to:agB
-node node2
  -pylon webSocket:pylon2 connectTo:ws://localhost:8886
    -agent composite:agB -shard messaging -shard PingBackTest
```

Fig. 1. Text-based deployment configuration for a simple scenario with two agents interacting via a WebSocket infrastructure. Each node contains one pylon of the interaction infrastructure and one agent.

```
-load_order driver;pylon;agent
-package ml aifolk.core aifolk.onto -loader agent:composite
-node node1 -driver ML:mldriver -driver Scenario:scen script:scr
  -driver Ontology:ont load:aifolk.ttl
-pylon local:pylon1
  -agent agA in-context-of:ML:mldriver in-context-of:Scenario:scen
    -shard messaging -shard scenario
    -shard MLPipeline -shard MLManagement feature:AutoDrive
```

Fig. 2. Deployment configuration for a scenario in which an agent interoperates with ML tools, via the *ML driver* entity.

P1: Any persistent component in a deployment is an *entity*, with a uniform model and that can be loaded in a similar manner, and is a first-class abstraction in the system. Entities include agents, artifacts, services, nodes, communication infrastructures, individual agent functionalities (which we model as *shards*), and so on. New entity types can be created and integrated seamlessly.

P2: Entities are linked by *context* and any entity can exist in the context of another entity. The relation does not necessarily make for a total ordering. For instance, agents exist in the context of nodes but also of communication infrastructures, which are embodied on a node by a *pylon*. Pylons run in the context of nodes, but, to be able to interact with other entities, nodes are also in the context of pylons.

P3: All entities are dynamically loaded and the implementation of an entity can be changed (a different class can be used) without impacting any other entity. For instance, the communication infrastructure can be changed without impacting entities which use it, and entities are ignorant as to what communication protocol is being used to allow them to interact.

P4: Entities interact via *wave*s, which transport *content* between *endpoints*. Any entity can have one or more endpoints and many entities are able to *route wave*s to other entities which exist in their context. E.g. pylons route *wave*s to agents and nodes, agents route *wave*s to their internal components, and so on.

P5: The motto of the project is "Easy for beginners, powerful to experts", with the target that running a simple scenario should be very easy, but deploying a complex scenario should also be possible, the complexity of specifying the deployment mirroring the complexity of the scenario itself.

In the following sections we will present some recently developed features of FLASH-MAS, which highlight the flexibility and versatility of the framework.

2 Integrating New Entity Types

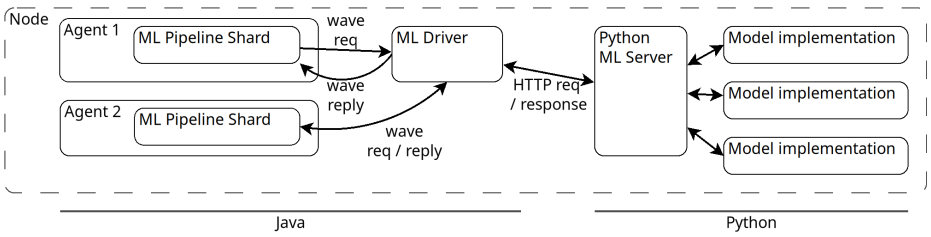


Fig. 3. Integration of Java agents with prediction models in Python, via a local web server.

Deployment of a FLASH-MAS scenario can be specified via XML files, but it is much easier to just use the command line. Running two agents[2] can be as simple as

```
java quick.Boot -package testing -agent PingPong:AgentA sendTo:AgentB
                                 -agent PingPong:AgentB
```

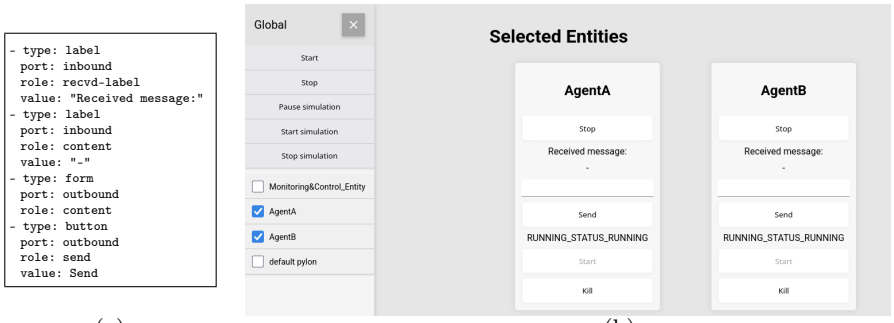
The framework automatically creates a node and a default communication platform, and automatically searches for the specified classpaths in the default and in the specified packages.

Adding a custom agent type and a specific communication platform is easy thanks to specifying custom loaders, automatic context inclusion (based on CLI parsing) and automatic location of classes – see Figure 1.

New types of entities can be added and loaded automatically (by specifying the `load_order`), and custom context relations can be specified explicitly using the `in-context-of` parameter for entities. In the next section, we elaborate on the deployment of a scenario in the AI Folk project, the deployment configuration of which is given in Figure 2.

In this scenario, the *driver* is a custom entity type, which is integrated seamlessly in the system at deployment time, without entities such as nodes knowing anything special about them – only the entities which need that functionality, and which are placed *in the context* of the driver, are concerned about the existence of that driver. Using the `load_order` element in the deployment configuration, one can specify what type of entities should be automatically loaded by the framework in each node.

[2] We omit the classpath parameter (`-cp`) of `java` for space reasons, and also for readability, but only one or two paths are actually needed, depending on whether FLASH-MAS is run from a `.jar` or using classes and dependency libraries.



(a) (b)

Fig. 4. Remote visualization interface built from a simple YAML specification (a) and presented in a web interface (b).

3 Interoperating with AI tools

We have been experimenting with integrating machine learning (ML) tools with FLASH-MAS and in the AI Folk project [7] we have developed agents which automatically select the prediction model to use in a task. Thanks to the flexible structure of FLASH-MAS, new entities are integrated easily (see the previous section), but to run prediction models implemented in Python a cross-platform, asynchronous interface with the Python language needed to be developed.

The link between languages was done via a local Python web server that receives requests from the `MLDriver` entity, which routes requests from all entities on that node. In the spirit of principle P3, prediction models are dynamically loaded via a configuration file. The interaction is shown in Figure 3.

The local web server offers, via the *ML driver*, functionality to

- load a new prediction model from a file containing the parameters;
- execute training of a model on a given set of examples;
- retreive model information;
- save the parameters of a model to a file;
- make a prediction using a model, on a given input, and return the result;
- list the available prediction models;

Since prediction models come in great variety and each one has its own mechanisms for loading, training, encoding input, and processing output, we abstract these processes using a Python class for each type of model, which extends the `ModelOperations` abstract class, and which offers a uniform API for loading, saving, prediction, training, and input and output processing.

4 Visualizing and controlling entities remotely

Visualization in a MAS is important. We have been working on a feature to allow entities (agents, especially) to present automatically generated interfaces based

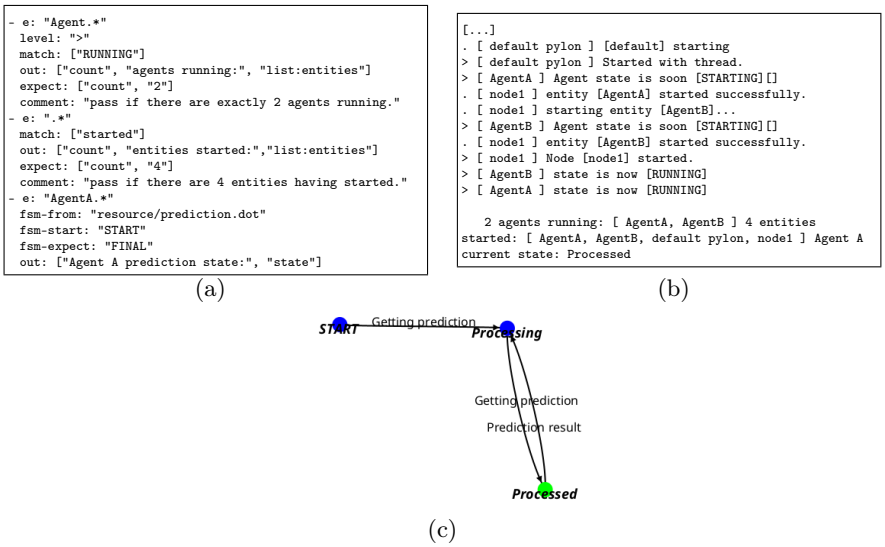


Fig. 5. Log summarizing using the `easyLog` tool: (a) the configuration specifies rules for what is matched, what should be displayed, and what is expected; (b) a summary line is kept at the bottom of the console aggregating outputs for each configuration entry; (c) a graph is displayed with the current state of each FSM.

on a list of input and output "*ports*" through which an entity can receive input or display output. The interface of the agent can be displayed locally as a Java Swing window but also remotely in a web interface that gathers information from all entities in a deployment and allows the user to interact with them.

Figure 4 shows the web application for viewing and integrating with agents. The interaction is possible via a *remote* agent shard which builds a description of the interface – fields, buttons, behaviors – from a YAML file (Figure 4 (a)) and interacts with a centralized *Monitoring Entity* to which it sends updates, and from which it receives updates, using normal FLASH-MAS communication via *wave*s. The Monitoring Entity creates a local web server which builds a global web interface based on the specification – see Figure 4 (b).

For agents containing multiple *shards*, each shard can contribute its own *ports* – hence its own controls – to the specification. The specification of the agent is combined with default controls, added by the Monitoring Entity, that show the status of the agent and buttons to start and to kill the agent.

5 Monitoring the progress of the system

Monitoring the state of a distributed MAS can be difficult when the number of entities is large and the scenario is complex. We have developed and integrated in FLASH-MAS a tool for tracking and summarizing the state of the system based solely on log entries produced by the entities. Provided all log entries

are captured in a single stream (e.g. the *standard output stream*), the `easyLog` tool provides a summary of current state. The configuration is very simple – see Figure 5 (a) – and specifies what log lines should be matched, what should be the output aggregating those lines and what are the satisfiability criteria. Output is provided as a line staying at the bottom of the console, by simply matching log lines with regular expressions, or by running a finite state machine (FSM) which changes state when certain matches are found in log lines for a given entity (see Figure 5 (b-c)).

6 Conclusion and Future Work

The Flash-mas framework is evolving continuously and the tools for deployment, integration, and monitoring of the system become more capable and easier to use.

Future developments include a documentation effort which is currently in progress and improving tools for automatic deployment on multiple nodes and more powerful state-based monitoring, together with interface and monitoring aggregation across larger numbers of agents.

Many thanks go to the many contributors to the Flash-mas project, listed in the `CONTRIBUTORS.md` file in the project repository, and especially to Ștefan Sterea, Tudor Ciot, and Valentin Mignot who have had important contributions to implementation of the features presented in this paper.

Acknowledgments. This paper is supported by the European Union's HORIZON Research and Innovation Programme under grant agreement No 101120657, project ENFIELD (European Lighthouse to Manifest Trustworthy and Green AI).

References

1. Agent2Agent Protocol (A2A) , https://google-a2a.github.io/A2A, (accessed 2025)
2. Model Context Protocol, https://modelcontextprotocol.io, (accessed 2025)
3. Bellifemine, F., Poggi, A., Rimassa, G.: JADE - a FIPA-compliant agent framework. In: Proceedings of PAAM. vol. 99, pp. 97–108. Citeseer (1999)
4. Collier, N.: Repast: An extensible framework for agent simulation. The University of Chicago Social Science Research **36**, 2003 (2003)
5. FIPA: FIPA abstract architecture specification (December 2002), http://fipa.org/specs/fipa00001/SC00001L.html
6. Olaru, A., Sorici, A., Florea, A.M.: A flexible and lightweight agent deployment architecture. In: 2019 22nd International Conference on Control Systems and Computer Science (CSCS), Bucharest, Romania, 28-30 May 2019. pp. 251–258. IEEE (2019). https://doi.org/10.1109/CSCS.2019.00048, https://ieeexplore.ieee.org/abstract/document/8744845/
7. Olaru, A., Sorici, A., Nan, M., Iancu, D.T.: AI Folk: Sharing machine learning models in a multi-agent community. In: Chinthaginjala, R., Sitek, P., Min-Allah, N., Matsui, K., Ossowski, S., Rodríguez, S. (eds.) Distributed Computing and Artificial Intelligence, 21st International Conference. pp. 119–128. Springer Nature Switzerland, Cham (2025)

8. Palanca, J., Terrasa, A., Julian, V., Carrascosa, C.: Spade 3: Supporting the new generation of multi-agent systems. IEEE Access **8**, 182537–182549 (2020)