The Thirty-Third International
FLAIRS Conference (FLAIRS-33)

Middleware Unifying Framework for Independent Nodes System (MUFFINS)

James S. Okolica,[1] Gilbert L. Peterson,[1] Michael J. Mendenhall[2]
[1]Department of Electrical and Computer Engineering, Air Force Institute of Technology, WPAFB, OH, 45433
[2]Human Performance Wing, Air Force Research Laboratory, WPAFB, OH, 45433 *

Abstract

Multi-agent systems are used in domains where individual component autonomy and cooperation are necessary. The overall system performance requires that the diverse agents maintain quality interactions to facilitate cooperation. A complication to inter-agent interaction occurs when the agents learn (change their own functionality), when new agents are introduced, or existing agents are functionally modified. This research focuses on creating a general use multi-agent system, Middleware Unifying Framework For Independent Nodes System (MUFFINS), and implementing a mechanism, the Megagent, that addresses the interaction challenges. The Megagent provides the ability for agents to assess their performance per data source and to improve it with transformations based on feedback. Evaluation of the concept is tested on data mangled from the Digits dataset to represent learning and new agents and in all cases improves accuracy over a static agent.

Introduction

Multi-agent systems (MAS) are "systems composed of multiple, interacting computing elements known as agents" (Wooldridge 2009). Two key aspects of agents are that they are capable of acting at least somewhat autonomously and that they are capable of interacting with other agents. The interaction requires an agent to reason over another agent's capabilities. This can be handled during the design time of the system (Darimont et al. 2016) or during continuous integration and testing (Nguyen, Perini, and Tonella 2007). However, if the agents are within an executing system and modifying their responses, or the system has no continuous integration test framework, the agents need to address the reasoning issue themselves. Rather than levy a requirement that every agent in a multi-agent system be able to perform on-line learning, we address this learning problem within the MAS implementation itself.

This work presents a mechanism, the Megagent, that enables agents to assess their performance on a per data source basis and improve it using feedback received from downstream agents. Evaluation of the Megagent's ability to learn about changes in their publisher agents is conducted with a set of seven agents that perform classification on the Digits dataset (Alimoglu 1996). To simulate the changes that can occur, there are data manglers that change the distributions of the features to simulate an agent that has learned a new representation. The remaining five data manglers perform drops or swaps of the columns that represent new agents that are added to the system that say that they provide the same vector that the `digitsClassifier` agent works on, but don't fully match the anticipated format.

Results demonstrate that adding the Megagent to a MAS allows for agents that autonomously learn and provides robustness to new and changing agents. In two cases, the Megagent was able to match the unmodified accuracy. As the amount of change increases, it was able to maintain a reasonable accuracy and in all cases improve accuracy of the agent over situations in which it was not present.

Related Work

There have been several efforts to implement frameworks to provide an underlying platform for multi-agent systems. The Java Agent Development Framework (JADE) (Bellifemine, Poggi, and Rimassa 1999) uses the FIPA specification as a starting point and provides a middleware for developing agents capable of interacting with each other. It does this by creating a platform (collection) of containers (agents) and hiding the underlying communication complexities from the developers. Zeus (Nwana et al. 1999) is a toolkit for developing multi-agent systems. In addition to providing an underlying communication channel, it also provides mechanisms for developing agents, including ontologies so that agents can understand each other, and predefined organization relationships so that agents can be organization and coordination can be defined. While this works well for when the tasks are known beforehand, it is too rigid for tasks defined after the agents are already well-defined.

The MUFFINS Framework

To provide an experimental testbed for evaluating the Megagent concept as well as future work, we have developed

*The views expressed in this paper are those of the authors and do not represent the views or policies of the United States Air Force or the Department of Defense.

Middleware Unifying Framework For Independent Nodes System (MUFFINS). MUFFINS is a MAS that provides agents coordination autonomy. Agents optionally accept input and provide output. Each agent definition provides pre- and post-conditions (similar to the preconditions and effects defined in the task definitions of Zeus (Nwana et al. 1999)) to describe the input and output they accept and produce. MUFFINS uses the Planning Domain Definition Language (PDDL) (Ghallab et al. 1998) to define the pre- and post-conditions. PDDL was selected based on its wide acceptance and the expectation that future extentions to MUFFINS could include a planner agent that optimizes connectivity. Figure 1 shows an example that consists of a single generic vector data object and several agents.

```
Vector Data Object Definition
(define (:domain Vector)
   (:types
      DataType - object)
   (:constants
      Real Integer - DataType)
   (:functions
      (dimension ?vector - Vector)
   )
   (:predicates
      (featureType ?vector - Vector ?x - DataType)
            (feedBack ?vector - Vector)
   )
)
```

```
(define (agent CSVProvider) (:domain Vector)
   :parameters (?output – Vector)
   :precondition (false)
   :postcondition (and (assign (dimension ?output) DIMENSION)
                       (featureType ?output DATATYPE)) )
```

```
(define (agent WekaClassifierAgent) (:domain Vector)
   :parameters (?input ?output - Vector)
   :precondition (or (and (>= (dimension ?input) 5)
                          (<= (dimension ?input) 30)
                          (featureType ?input Integer))
                     (feedback ?input))
   :postcondition (and (assign (dimension ?output) 1)
                       (featureType ?output Integer))
```

```
(define (agent Analyzer) (:domain Vector)
   :parameters (?input ?output – Vector)
   :precondition (and (assign (dimension ?input) 16)
                      (featureType ?input Integer))
   :postcondition (and (assign (dimension ?output) 1)
                       (featureType ?output Integer)))
```

Figure 1: PDDL Definitions.

MUFFINS uses a publish/subscribe mechanic for inter-agent coordination and communication. The process of creating subscriptions is shown in Figure 2. When a new agent enters the network, it broadcasts an ENTER message listing its outputs (step 1). When existing agents see the new agent, they send it an ENTER message listing their outputs (step 1). As illustrated in step 2, as agents receive lists of outputs from other agents, they evaluate whether their inputs are compatible with a subset of the outputs other agents produce. For instance, a source agent may produce vectors of dimension 1 and vectors of dimension 16. If a transformer agent takes vectors of dimension 16 as input, they are compatible with that source. As shown in step 3, when an agent discovers an agent with compatible outputs, it sends a subscription request to that agent along with its input. The potential publisher agent checks the inputs with its outputs and, as seen in steps 4 and 5, if it finds a match (i.e., a set of its outputs that matches a set of the potential subscriber's inputs), it adds the subscriber to one of its publication groups. If the subscriber's inputs match multiple outputs for the publisher, it adds the subscriber to multiple publication groups. Finally, as shown in step 6, the publisher sends a confirmation to the subscriber for each group it has joined it to.

As shown in Figure 3, with the inter-agent communication handled in the `lowerLevelCommunicationLayer` and the Subscription process handled in the `nodeCommunicationLayer`, the abstract class `agentBase` is defined independent of these imple-

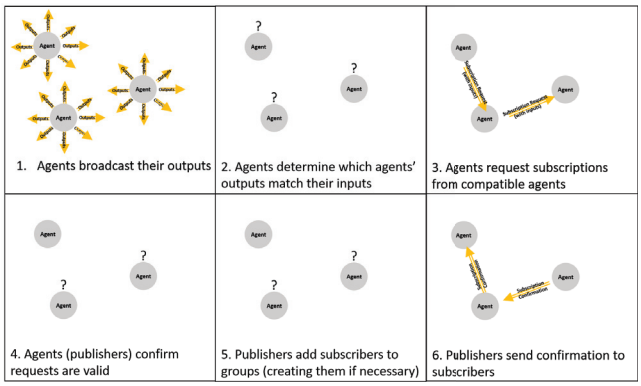

Figure 2: PDDL Subscription Phases.

mentation details. There are five methods that all agents perform and are implemented in the `agentBase` class: providing their unique unit identifier (UUID), next job id and sequence number, sending data to other agents either via publication groups or peer-to-peer, spawning other agents, processing data, and, for source agents, starting ancillary threads to send data.

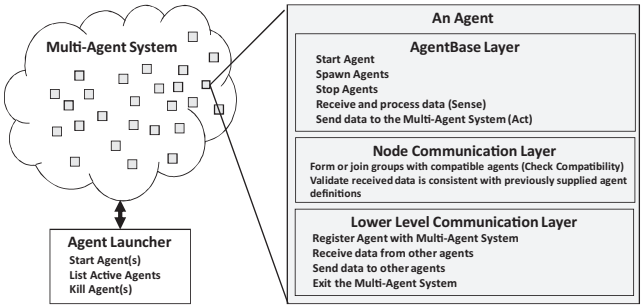

Figure 3: MUFFINS Framework.

Message types are implemented using Google's "language-neutral, platform-neutral" protobuf protocol buffers for serializing structured data (Google 2018). There are ten message types: two for subscription processing (ConnectionRequest and ConnectionResponse), two for the health monitor system (TransactionOverviewRequest and TransactionOverviewResponse), one for passing UUIDs from spawned processes to their parents (SpawnAgentResponse), one feedback message type (Feedback) and four data message types (Vector, ImageData, PythonPickle, ProtoDataFrame). Several of the message types (e.g., the ones for subscription processing, spawning and transaction overview requests) are handled by either the `nodeCommunicationLayer` or `lowerLevelCommunicationLayer` and never get to the `agentBaseLayer`, leaving the agent responsible for no more than the transaction overview and four data message types.

The Megagent

While the MUFFINS framework addresses an assembly of autonomous agents discovering and connecting to each other, it does not address the central question of enabling agents' ability to reason over other agents during execution time. While it is possible to put the burden for this learning on individual agents, it is more efficient to create a mechanism within the MUFFINS framework, making it possible for agents to include it as needed. The first step is adding feedback to MUFFINS. As agents assess each other's performance, they can decide whose output is 'good enough', a user-specified and task-specific value, for them to use as input. Furthermore, as agents receive feedback on the output they are producing, they can use that feedback to improve their performance.

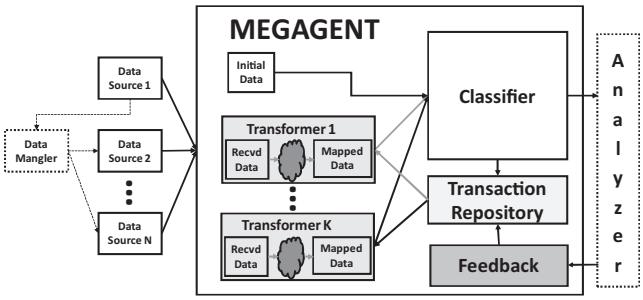

Figure 4: Megagent Architecture.

Figure 4 provides an overview of a Megagent with a static classifier (the dashed boxes and arrows are used for testing). Initially, there are no transformers. The Megagent receives data from different data sources, sends them to the Classifier, stores the input and results in the Transaction Repository and sends results as output. At some future time, the Megagent receives feedback on a result. The feedback may come from one of the agents it sends its output to or from an external agent (possibly an agent further downstream, possibly an agent "listening in"). The feedback references a specific job that the Megagent performed. The Megagent then associates the feedback with its initial classification for the job and stores both in the transaction repository. If the Megagent's performance feedback on data from a specific data source drops below a threshold level, a user-specified and task-specific value, it then uses the transactions to build a Transformer agent. In the future, all data received from that data source will go through a transformer associated with that data source. If the cumulative accuracy drops below the threshold a second time, the Megagent determines its performance is unacceptable and stops accepting input from that data source.

When the transformer instantiates, it begins by taking the training data passed to it by the Megagent and creating multiple candidate transformations to determine which performs best. To accomplish this, it uses a dedicated communication channel established with the Megagent's primary agent (e.g., a classifier). For each transformation, it takes each exemplar, transforms it and sends it to the primary agent for processing. The primary agent processes it and sends back the result. It takes the result and compares it to its truthed training data. After each transformation has processed all of the training data, the `transformer` class chooses the best transformation and informs the Megagent it is now started and ready.

Currently, the `transformer` class includes four types of transformations: a linear transformation, a k-means clustering (MacQueen and others 1967) algorithm using statistical moments (mean, standard deviation, skew, and kertosis), a locality sensitive hashing (LSH) (Indyk and Motwani 1998) algorithm and a combination of the k-means clustering and LSH algorithms. The linear transformation simply builds a linear transformation using the training data and a Euclidean distance calculation. The goal of the k-means clustering is to determine which feature a given feature actually is. Consider the vector $(1, 2, 3, 4, 5)$. Does the first feature (value 1) actually represent feature one or does it actually represent feature three? The k-means clustering begins by creating a single cluster for each feature in each class using the statistical moments of the data the primary agent was originally trained on. It then adds additional clusters for features until it drops the average intra-cluster distance below a user specified threshold (12.5 for the experiment). It then creates a transformation using the training data from the statistical moments of the data source in question. Locality sensitive hashing (LSH) creates hashes using subsets of the features to produce a representative exemplar for a class. The size of the subsets is based on a user specified value (6 for this experiment). The final transformation first performs the k-means clustering to swap the features as needed and then uses LSH to discover a class-representative exemplar.

Experimental Evaluation and Results

Initial testing of the Megagent was done using the Digits dataset (Alimoglu 1996) which consists of approximately 7500 training and 3500 test handwriting samples. The preprocessing converts raw data from a pressure sensitive tablet into fixed length feature vectors of sixteen integers ranging in value from zero to one hundred.

Figure 4 shows the architecture for the experiment. The base agent is a J4.8 (Frank, Hall, and Witten 2016) decision tree classifier trained on the training data. The decision tree has an accuracy of 91%. Then two data sources are created. The first outputs the original test data. The second outputs the test data mangled in one of seven ways. The Megagent-wrapped classifier receives the source data, classifies it and sends it along to an analyzer. The analyzer then provides feedback back to the classifier consisting of the correct answer. For this experiment, the megagent triggers creation of a transformer when the accuracy for a single data source is below 70% and it has processed at least 500 transactions.

Table 1 shows results from the seven tests. In the first two tests, the test data was perturbed from its true value by a random amount using a normal distribution with a standard deviation of 10 and 16 respectively. This represents an agent that may be learning or alternatively an agent that has a degrading sensor. The next test is a shuffling of the features which represents a new or revised agent that is sending the

Table 1: Megagent Results.

Input Data Manipulation	Pre-Transform Accuracy	Post-Transform Accuracy	Identified Transformer	Percentage Improvement
Noise (std dev 10)	0.70	0.89	Linear Transform	27
Noise (std dev 16)	0.56	0.75	Linear Transform	34
Shuffle order (20 swaps)	0.13	0.91	Statistics	600
Zero 6 of 16 columns	0.43	0.83	Locality Sensitive Hashing	93
Zero 8 of 16 columns	0.30	0.77	Locality Sensitive Hashing	157
Zero 6 of 16 columns and shift left remainder	0.11	0.81	Locality Sensitive Hashing and Statistics	636
Zero 6 of 16 columns and shuffle order (20 swaps)	0.12	0.69	Locality Sensitive Hashing and Statistics	475
Baseline		0.91		

same data just in a different order. Tests four and five involve zeroing out either 6 or 8 of the features. These tests represent a new agent that may not provide all of the functionality of the original. In the final two tests, in addition to zeroing out some of the features, the remaining data is either compressed or shuffled as above. The results are shown in Table 1. In all but one case, when the Megagent added a transformer, the resulting accuracy improved above the threshold. In the final case, the results were just below the threshold. The increase in accuracy ranged from 27% for minimal data changes to a high of over 600% when the input is significantly modified.

Conclusions and Future Work

This paper presented the Megagent concept in which the ability to handle autonomously learning agents in a MAS is integrated into the MAS infrastructure. This provides each agent the capability to represent and reason about the actions and knowledge of other agents. The Megagent was demonstrated in a MAS with the task of performing hand written digit identification. Experimental results show the Megagent dramatically overcoming data mangling based on feedback received. MUFFINS and the Megagent allow for including static agents when existing agents can change their knowledge and actions or in cases when new agents are introduced into existing systems.

For this experiment, feedback was provided immediately; it was provided for each exemplar; and feedback consisted of truthed data. Future work should handle and evaluate intermittent feedback which provides a score of how good the answer is (e.g., real valued feedback between 0.0 and 1.0). Furthermore, while the Megagent provides a first step toward self-assembly, it is only a first step. Agents in the multi-agent system also need to have some concept of task, how well they perform on different tasks and how good the input is from different other agents for different tasks. With this additional information, self-assembly in multi-agent systems will become closer to reality.

References

Alimoglu, F. 1996. Combining multiple classifiers for pen-based handwritten digit recognition. Master's thesis, Institute of Graduate Studies in Science and Engineering, Bogazici University.

Bellifemine, F.; Poggi, A.; and Rimassa, G. 1999. JADE–a FIPA-compliant agent framework. In *Proceedings of Practical Applications of Intelligent Agents*, number 97-108 in 99, 33. London.

Darimont, R.; Zhao, W.; Ponsard, C.; and Michot, A. 2016. A modulare requirements engineernig framework for wbe-based toolchain integration. In *2016 IEEE 24th International Requirements Engineering Conference*, 405–406.

Frank, E.; Hall, M. A.; and Witten, I. H. 2016. The WEKA workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques".

Ghallab, M.; Howe, A.; Knoblock, C.; McDemott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL - the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Google. 2018. Protocol buffers. https://developers.google.com/protocol-buffers/.

Indyk, P., and Motwani, R. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 604–613. ACM.

MacQueen, J., et al. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, 281–297. Oakland, CA, USA.

Nguyen, C.; Perini, A.; and Tonella, P. 2007. Automated continuous testing of multi-agent systems. 1–19.

Nwana, H. S.; Ndumu, D. T.; Lee, L. C.; and Collis, J. C. 1999. Zeus: a toolkit for building distributed multiagent systems. *Applied Artificial Intelligence* 13(1-2):129–185.

Wooldridge, M. 2009. *An Introduction to Multiagent Systems*. John Wiley & Sons.