USING META-AGENTS TO BUILD MAS PLATFORMS AND MIDDLEWARE

S. C. Lynch
School of Computing, Teesside University, Middlesbrough, UK
s.c.lynch@tees.ac.uk

Keywords: Meta-agents, platforms, middleware, distributed systems.

Abstract: Various multiagent platforms exist, each providing a range of individual capabilities but typically their implementations lack the flexibility to allow developers to adapt them to the differing needs of individual applications.

This paper investigates the design of a kernel for MAS middleware based on primitive meta-agents. We specify these meta-agents and examine how they can be used to realise the capabilities required by multiagent platforms. We examine how different configurations of meta-agents support distributed systems of application-level agents and consider how changes in the organisation of meta-agents produce MAS platforms with differing behaviours. We demonstrate the use of our design by building a set of meta-agents in Java and evaluate the meta-agent approach by experimentation, demonstrating how modifications in meta-agent behaviour can provide different strategies for agent communication, scoping rules and connectivity with other tools.

1 INTRODUCTION

Many MAS platforms are currently available to developers, these platforms vary in their range of capabilities, the facilities they offer and their notions of agency. Some platforms concentrate on support for BDI agents (Mascardi et al, 2005), others on mobility (Cabri et al, 2006; Suna & Fallah-Seghrouchni, 2005), others intend to be more general purpose (Bellifemine et al, 2008; Cheyer & Martin, 2001). Some platforms provide programmer support in the form of debuggers and developer tools, others do not (Bordini et al, 2006). With the exception of MadKit (Gutknecht & Ferber, 2000), none of the platforms we surveyed allowed system developers to modify the underlying behaviour of the platform or adapt the functions of its middleware. This inflexibility has been noted by other authors (eg: Fonseca, 2006) who have highlighted that frameworks "*support their own specific flavour of agency (generally emphasizing characteristics that are inline with an anticipated application domain or research interest) that cannot be readily adapted*" (Fonseca, 2006).

Lack of adaptability is not limited to agent platforms but is common throughout the history of computing artefacts where developers have been expected to build systems using languages and subsystems with well-defined but immutable behaviour. For programmers this has meant that the semantics of the languages they use are fixed and, while they can build functionality on top of these languages, they cannot modify the language itself – even in situations where adjustment of a language's semantics would permit the development of more elegant solutions. For MAS developers it means that they can build systems on top of platforms but in doing so must work within the constraints imposed by the platform they are using.

The introduction of the metaobject protocol sought to improve the situation for object programmers: "*Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behaviour and implementation, as well as the ability to write programs within the language*" (Kiczales, 1999). Despite its benefits, the adoption and use of metaobject protocols has been modest in comparison to other developments in language design. This may be because specifying semantics for programming

languages is highly complex and beyond the capabilities of many programmers, there is no reason, however, to presuppose that these difficulties would be equally present at the agent level of abstraction.

Other authors have suggested approaches which could be used to provide analogous facilities for MAS. Kind and Padget suggested meta-agents (Kind & Padget, 1999) and others have discussed the use of meta-actors for middleware (Sen & Agha, 2002). This work may have influenced the design of some MAS platforms (Mulet et al, 2006) but meta-agent design patterns are not an area for significant investigation in the agent research community, there has been little discussion of the way meta-agents are constructed and the ways they may be reconfigured to provide multiagent platforms and middleware with differing capabilities. Not only that but MAS platforms do not present their users with a meta-agent layer but instead provide a rigid structure for their middleware which imposes various limitations and predetermines (i) the system architecture – hub-based or peer-to-peer for example (ii) in some cases the nature of channels for inter-agent communication (sockets, CORBA, etc) (iii) the scale of platform's middleware – typically, the same middleware is provided in all situations. This prevents, for example, light-weight versions of middleware being used in circumstances where this would be preferred such as with constrained mobile devices. This can also be disadvantageous for developers in various other situations. 2APL (Dastani, 2008) for example uses JADE to provide base agency and middleware when 2APL solutions are distributed across VMs but implements its own message transport service when applications exist in a single VM, the rationale for this is that JADE provides more than is required for simple message passing within a single VM.

MadKit was presented as an exception to this norm, built from a kernel of low-level agents, it aimed to provide a extensible platform which could be customised by its users (Gutknecht & Ferber, 2000). MadKit offers some flexibility but has not been widely adopted as a base-layer or middleware solution (2APL, Jason and Goal all use JADE for example). Consideration of MadKit suggests that the use of a meta-agent micro-kernel is worth further study.

Our main aim is to use configurations of meta-agents to support distributed systems of application-level agents and to consider how changing the organisation of these meta-agents produces MAS platforms with differing behaviours. To this end we investigate using meta-agents to provide a generic but extensible middleware which may be reconfigured to suit the requirements of individual applications or their deployment needs. Specifically we wish to address the following:

- generality: platforms should not have pre-existing notions of applications or specialised concepts associated with the base technology like specific variants of agency;
- flexibility: platforms should facilitate building simple, light weight systems with minimal features as well as more complex systems;
- ease of translation: a design should lend itself to implementation in a variety of programming languages;
- to provide a flexible underlying architecture which can be used for hub-based, peer-to-peer or hybrid architectures;
- extensibility;
- to facilitate interconnectivity with other systems.

This paper presents a design for MAS middleware which addresses these aims. We build a small set of primitive meta-agents and show how they can be organised into different configurations. These configurations, which are themselves distributed systems of meta-agents, form platforms to support application-level agents. The differences in configurations produce adaptations in the characteristics of the underlying systems which change the behaviour of applications built on them.

We show how meta-agents can be constructed from a small set of language-level primitives and investigate how they can be used to support strategies for communication between application agents in a distributed environment as well as the behaviour of those agents. The resulting meta-agent configurations provide adaptable middleware solutions for application MAS. The design lends itself for implementation in various languages and permits easy connectivity with other systems, we have successfully used the design to build platforms in Java, Lisp and C# and have developed links to Galaxy, .NET and JADE.

2 A META-AGENT SUBSYSTEM

Conceptually each application agent rests on a small set of interconnected and interacting meta-agents which provide the application agent with its abilities to encapsulate behaviour and to communicate. In a distributed environment agents may reside on different physical or virtual machines. Our aim is for meta-agents to be light-weight and offer maximum

opportunity for reuse and reconfiguration so we separate the functions of behaviour and communication into different meta-agent classes. In the following discussion a *Portal* is defined as a specialisation of meta-agent which provides communication between agents and a *socket-agent* is another specialisation of meta-agent which routes messages between physical or virtual machines through a socket. Fig.1 shows the structure for this arrangement with two application agents A1 and A2 (the meta-agents named ma1 and ma2 provide the behaviour for A1 and A2).

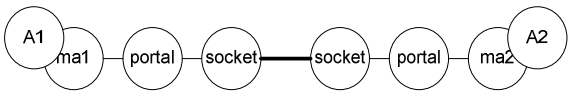

Figure 1: A simple meta-agent structure.

For the purposes of our discussion here we consider meta-agents to be primitive meta-agents, small autonomous software entities with the following characteristics (i) they operate in their own process thread and can thereby be concurrently active with other meta-agents (ii) they have an inward communication stream capable of queuing incoming messages – this allows meta-agents to receive messages from other meta-agents (iii) they can encapsulate behaviour – including, as will be discussed later, behaviour which allows them to send massages to other meta-agents and to manage deliberation cycles (episodes of activity where meta-agents may be proactive rather than reactive).

The design of meta-agents is such that incoming messages are queued until their thread is idle then the least recent message is passed to the meta-agent message-handler and its thread is rescheduled. The message handler is used to encapsulate the behaviour which occurs in response to messages. Meta-agents are specified as objects so the nature of this behaviour is not restricted and may involve modification of instance or environmental data.

Other details concerning the structure of agents, the functions they may perform and the nature of their inter-agent communication is left unrestricted.

2.1 Meta-Agent Specification

In this section we specify types of meta-agent and consider ways in which they may interact to produce communicative behaviour. We specify these in a notation which makes object classes and their methods explicit and also the multi-threaded behaviour of their operation. Initially a meta-agent can be specified by extending the concept of a *blocking queue*. A blocking queue behaves like a typical FIFO queue except that if data is requested from it when it is empty, it suspends the thread of the process requesting data until the queue becomes non-empty (ie: until some other process pushes or enqueues new data). Assuming the class *blocking-queue* exists with the capability to *enqueue* and *dequeue* data, meta-agents can be specified as shown below. Instances of meta-agent have a name and a link to a *portal*.

```
class meta-agent
  extension-of: blocking-queue
  variables: name, portal

  constructor method( args... )
  | super.constructor( args... )
  | thread{ loop-forever
  |    msg-handler( dequeue() )}

  method msg-handler( msg )
  | ;; reactive behaviour to messages
```

The construction/instantiation of a meta-agent creates a new process thread for the agent which continually extracts messages from the agent's queue and passes them to the agent's message handler, the reactive behaviour for responding to messages is specified in the message handler. The use of a single thread for an agent ensures that messages it receives are processed sequentially. This is often the type of behaviour expected/desired of agents but not in all cases. In other situations it may preferred that as new messages are received, new run-time instances of agents are produced to process these new messages. This type of behavior allows multiple clones/duplicates of an agent to exist simultaneously. Often when authors refer to cloning/duplication they are discussing a complex process with a strategic underpinning (Fedoruk & Deters, 2002), in the case of meta-agents we use the term clone/non-clone only to imply a simple modification to behavior. Some meta-agents will only ever process messages sequentially (non-clones) while others (clones) may process them concurrently. To allow cloneable behavior the meta-agent constructor method can be redefined to operate in a new thread as follows:

```
meta-agent.constructor method(args...)
|   super.constructor( args... )
|   thread{ loop-forever
|      thread{msg-handler(dequeue())}}
```

2.2 Agents for Communication

This section considers the communication between meta-agents, communication schemes for application level agents are constructed using meta-

agent communication as a base layer. Portals manage communication for other meta-agents so when one agent sends a message to another it is sent via the sending agent's portal. The message contains the name of the agent which is to be its final recipient. The agent's send-message method handles this as follows (note: the function *wrap* is used to pack the recipient name and the message into a single, faceted form – the mechanism for this and the resulting structure is discussed later).

```
meta-agent.send-message(recipient, msg)
| portal.enqueue(
|    wrap(name, recipient, msg)
```

A portal is a specialised meta-agent which routes messages between other meta-agents. For the sake of simplicity we assume here that all meta-agents have locally unique names and portals map these names onto agent instances in order to forward messages onto the appropriate queue for any agent (alternative arrangements of meta-agents can be used to produce different white/yellow pages systems but these are not discussed here). Using portals allows meta-agents to communicate with each other by names but removes the need for them to locate each other since this becomes the portal's responsibility. The simplest specification for portal is as follows:

```
class portal extension-of: meta-agent
 variables: routing-table

 method msg-handler(msg)
 | routing-table.get(
 |     recipient-of(msg)).enqueue(msg)

method add-agent( name, agent )
 | routing-table.set( name, agent )
```

A portal's routing table is a hashing structure which maps agent names onto their queues (or their agent instances).

The specification of portal allows multiple agents to share the same portal but since it also implicitly allows portals to connect to other portals (by virtue of them being defined as agents themselves) it also allows each agent to have its own dedicated portal (see Figure 2 where A1 and A2 are standard meta-agents and P1 and P2 are portals depicted with their routing tables, arrows indicate movement of message data between components).

This provides the flexibility to develop both peer-to-peer architectures for agent communication or hub-based architectures (or some hybrid approach). Unfortunately, the specification of portal imposes an unsatisfactory constraint on the relationship between agents and portals since, for the portal to access an instance of an agent, the agent instance must either exist in the same execution space as the portal or there must be some exchange of objects between execution spaces (here the term execution space means Lisp world, virtual machine (VM) or executing program).

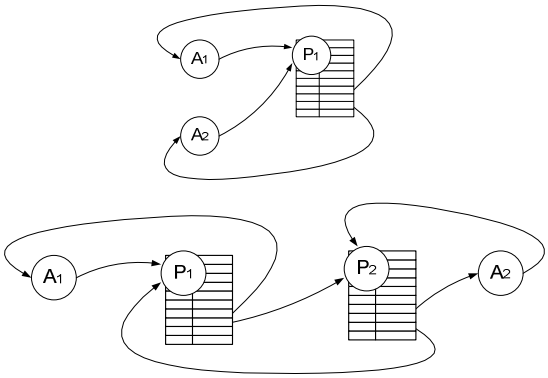

Figure 2: Agents sharing a single portal (top)and with independent portals (below).

We overcome this problem by introducing a third genre of meta-agent which is responsible for handling communication between execution spaces. In our implementations we have chosen to use sockets for this and name the new meta-agent a *socket-agent* (though other meta-agent definitions allow alternative communication channels to be used).

```
class socket-agent
  extension-of: meta-agent
  variables: socket

  method make-read-loop
  | thread{ while open(socket)
  |    portal.enqueue(
  |       text-to-msg( socket.read())

  method msg-handler(msg)
  | socket.write( msg-to-text(msg))
```

The process of setting up the socket for this agent requires contacting some target process to negotiate a connection which will allow the portal attached to this agent to reach some other portal via its socket agent. Once connection is established the socket agent has input and output streams, socket-agent.make-read-loop ensures that any data sent to it via its socket connection will be forwarded to its portal and the message handler writes messages to the socket connection (messages are converted to text for socket based transmission.

Since socket agents may be accessed by a portal's routing table (like any other kind of meta-agent) their specification now allows portals and the agents attached to them to communicate between execution spaces and between machines on a network (see

Figure 3 where S1 and S2 are socket agents and the heavy line shows socket based communication).

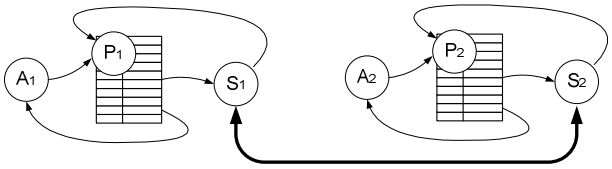

Figure 3: Use of socket agents.

2.3 Agent Registration

Agent registration is the process which deals with adding a new agent to the system. Given the model described above, three things must be achieved during registration: (i) the agent keeps a record of its portal (ii) the agent's portal adds the agent to its routing-table (iii) any other portals in the system are informed about the location of the agent. The location of an agent describes how it can be reached, portals make entries in their routing tables in order for messages to be correctly forwarded through the network of meta-agents. Note: this implies that all portals must hold routing entries for all agents, we will later outline alternative configurations of meta-agents which overcomes this requirement.

Message routing becomes increasingly indirect as the models progress from those shown in Figures 1 to 3. In Figure 1, the routing table of P1 maps A1's name to its instance. We write this as $A1 \rightarrow |A1|$ so in Figure 1, P1 has mappings as follows:

$$P_1 : \{ A_1 \rightarrow |A_1| , A_2 \rightarrow |A_2| \}$$

The portals in Figure 2 have the mappings:

$$P_1 : \{ A_1 \rightarrow |A_1| , A_2 \rightarrow |P_2| \}$$
$$P_2 : \{ A_1 \rightarrow |P_1| , A_2 \rightarrow |A_2| \}$$

In Figure 3 the mappings are:

$$P_1 : \{ A_1 \rightarrow |A_1| , A_2 \rightarrow |S_1| \}$$
$$P_2 : \{ A_1 \rightarrow |S_2| , A_2 \rightarrow |A_2| \}$$

These mappings are established when agents register and notification about their registration is propagated through the system. Using the model shown in Figure 3, consider the case where a new agent A3 registers using P1 as its portal, the following steps occur:

(i) P1 establishes the mapping $A3 \rightarrow |A3|$ in its routing table and forwards the mapping $A3 \rightarrow |P1|$ to S1;

(ii) S1 forwards the mapping $A3 \rightarrow S1$ to S2 (since S1 and S2 do not share execution space, instance data cannot be exchanged so only the name of S1 is passed);

(iii) S2 forwards the mapping to $A3 \rightarrow |S2|$ to P2;

(iv) P2 adds the $A3 \rightarrow |S2|$ mapping into its routing table.

There is an additional mechanism for sharing routing-table details between portals when two portals (which may already possess non-empty routing-tables) first connect to each other. In this situation when two subsystems link additional routing information is exchanged. Further specialisations of meta-agents can be used to construct naming services and the use of portals combined with scoping (discussed below) allows localisation of routing tables and varied strategies for message broadcasting.

2.4 Internal Message Protocols

In a completed platform meta-agents (portals, sockets, etc) transmit data relating to messages sent between user-agents as well as other types of transmissions intended only for other meta-agents (messages containing information about agent registration, etc). It is therefore necessary to have a protocol for internal data/messages which allows transmission of different types of information. In addition, since we wish to specify a design for a generic MAS platform that may be readily modified/extended, the internal message protocol must also be adaptable.

In the discussion above we have used *wrap* as a mechanism to pack different data into a single form for message transmission and accessor methods to unpack the data but have not examined how the data can be encoded. Here we propose a simple slot-filler notation with a Lisp-like textual form and an *event-type* slot which describes the overall type of the message. This type of information is *meta-data*, it is only used by meta-agents. Below is an example of a simple message which informs its recipient that an agent named *agent3* has registered with the system and provides appropriate mapping.

```
((event-type register)
 (name agent3)(map-to socket5))
```

Messages between user-agents will typically contain more data. Shown below is a representation of a message transmitted between two user agents *agent-sue* and *agent-kim* which contains information about the sender of the message as well as a unique identifier for the message.

```
((event-type user-agent-message)
 (body message-text) (sender agent-sue)
 (recipient agent-kim)(msg-id #2))
```

One of the benefits of a MAS approach is that MAS can be restructured by adding agents to provide additional functionality. Our aim is to construct a raw, generic design which can easily be extended to deal with specific issues of agency required by more specialist implementations (mobility for example). To avoid over-constraining the types of information which can be sent between meta-agents we permit additional event-types to be introduced and additional slots to be used with existing event-types as necessary. An example of this (used in the later section on scoping) is a form of registration message which contains scope information provided in an additional slot.

```
((event-type register) (name agent3)
 (scope global) (map-to  socket5))
```

2.5 External Environments

In addition to existing in a social environment of other agents, evidenced by the messages exchanged between them, agents often also exist in an external environment (mapping onto some physical or virtual world state). Some agent languages/platforms achieve this by providing specific syntax for sensing the status of agents' external environments and effecting/requesting changes to those environments.

To realize this behaviour, agents require the ability for (two-way) communication with their environments and/or some mechanism to respond to notifications of changes in those environments. This may be achieved with meta-agents by associating an *environmental meta-agent* with each environment or agent-environment link. These environmental meta-agents process *sense-environment* requests messages from other agents, deal requests to modify the environment and, for agents required to react to changes in the environment as they occur, the environmental meta-agents take responsibility for broadcasting *changed-environment* information.

Application agents may then interact with their environments by interacting with environment meta-agents and need no other specialist support mechanisms for this.

3 EVALUATION

Following our initial investigations, we have used meta-agents to rebuild pre-existing middleware and MAS platforms used in our institution, successfully developing meta-agent subsystems in Java, Lisp and C# which allow distributed agents specified in different languages to freely interact. These subsystems have been further developed to link to the Galaxy Communicator architecture .NET services and JADE and have been successfully used as stable platforms for larger scale MAS projects involving multimodal dialog and other areas of research. These applications are not discussed here since the primary concern of this paper is to investigate the extent to which MAS implemented on top of meta-agent subsystems can have their behavior tailored by reconfiguring and/or modifying the meta-agents themselves. The following subsections examine various ways that the meta-agent subsystems which support application agents can be adjusted and the impact of these changes on the MAS platform they define.

3.1 Application Agents

Our first series of experiments involves building a useable system on top of the meta-agent subsystem. Primarily this requires the specification of additional classes for user-agents and portals to include some utility methods to aid usability.

Our aim is to build provision for the kind of agents MAS application programmers will develop. For the sake of discussion we call these application agents *user-agents*. In their simplest form, instances of user-agent need not differ greatly from those of meta-agent. However we can provide a more convenient form for the user-agent message-receiver method by unpacking the facets of data a user-agent message contains. In a completed implementation a message between user-agents will include the message body, the name of the sender and possibly information concerning the message identity and session identity. These additional message facets are readily generated by originating meta-agents and/or portals and enable replies to be generated for specific messages and tracking of sessions/dialog.

User-agent can be specified as a specialisation of meta-agent with an overloaded message-received method. The specification below assumes accessor methods for the different facets of data contained in user-agent messages. The internal protocol for transmitting this type of data is outlined in the following section.

```
class name: user-agent
 extension-of: meta-agent
 method message-recieved( message )
 | message-received(
 |     get-sender( message ),
 |     get-recipient( message ),
 |     get-message-body( message ),
 |     get-message-id( message ),
 |     get-session-id( message ))
```

In Java, the means for specifying code to be triggered when messages are received by user-agents is made to resemble the standard Java mechanism for responding to events – this is done to improve usability. The Java class containing these modifications is called Agent and is an extension of the user-agent class specified earlier. Following these modifications, programmed Java agents can be defined as self contained classes (which extend the Agent class) or specified inline as follows:

```
Portal p = new Portal( portal-name );
Agent a = new Agent( agent-name );
a.addMessageListener(
  new MessageListener()
  { public void messageReceived(
      String from,String msg,MsgData d)
      {   ...code body...
      }
  });
p.addAgent( a );
```

The messageReceived method is called when an agent receives a message, the MsgData argument provides access to additional information like message identity and session identity. The addAgent method is used to trigger the process of agent registration and the propagation of information used to update the routing tables of portals. Sending messages (not shown) is achieved with a simple sendMessage method defined as part of the Agent class.

Similar forms exist for specifying application agents in C# and Lisp.

3.2 Strategies for Message Routing

In a distributed environment, exchanging messages between agents may also involve data transmission across different hardware devices. The interconnections between agents define some abstract virtual network which is supported, at another level, by a physical network topology. In addition to providing agent behaviors, any generic meta-agent subsystem needs to support a variety of strategies for message transmission. The ability to modify the underlying virtual network structure for agent messaging is of particular interest since the requirements for MAS deployed over long distance networks or on mobile devices are likely to be different to those deployed on a small local cluster of machines.

In order to evaluate our design strategy, we have experimented with the configuration of meta-agents to produce different strategies for organising messaging infrastructure:

(i) peer-to-peer, this strategy logically puts both the behavior and the management of communication inside the agents who then manage their own communication channels directly;

(ii) using some kind of hub architecture where behavior and communication are placed in different software components. With agents managing behavior and some other component(s) managing communication.

Implementation of a peer-to-peer strategy involves enforcing the condition that each agent has its own portal which is not shared directly with other agents. This can be readily achieved in two ways: (i) by simply disallowing portals to be shared (this provides the kind of topology shown in Figures 1 and 2) or (ii) by redefining user-agent as an extension of portal-agent instead of a direct descendant of meta-agent.

The use of portal meta-agents in the sub-system provides a partial hub architecture without any further modification. However, we have extended the notion of a hub to a more extreme form by producing a new agent called a router. In this experiment, each portal can connect to multiple agents but is only allowed a single external link via a socket agent and this socket must connect to a router. Routers only connect directly to portals or other routers, they do not connect to agents (see Figure 4 for an example architecture).

In this experimental topology both routers and the new specification for portals are specialisations of PortalAgent but some additional constraints are imposed on their behavior. As with the peer-to-peer system, the necessary modifications were readily achieved and the subsystem behaved properly.

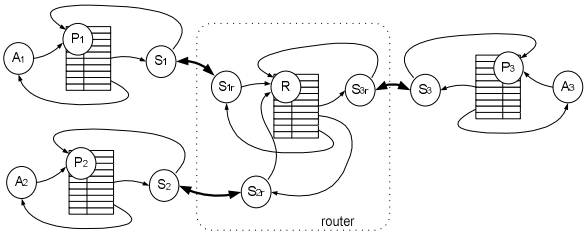

Figure 4: Hub architecture with router

We modified our new portals further to allow their agents to send messages to other agents which were not in their portal's routing table. In this scenario, internal agents (agents which connect directly to a given portal) have an entry in the portal's routing-table but external agents (agents not connected directly to a given portal) may not have an entry in its table. This was handled in a

satisfactory way by redefining the portal msg-handler as follows:

```
method portal.msg-handler (msg)
| target =
|  routing-table.get(recipient-of(msg))
| if target then target.enqueue( msg ))
|  else socket-agent.enqueue( msg ))
```

The socket-agent referred to in the code above is the portal's socket agent which connects it to a router. Unlike portals, routers are expected to have complete routing table information.

3.3 Scope Rules for Agents

A simple MAS may be composed of a homogeneous collection of interacting agents in which all elements of the system are visible to all others. Some MAS however, are not best organised as single collections. In the case of large MAS, there may be benefits in grouping clusters of agents which, for many purposes, can be treated as single entities. These may relate to organisations (Massonet et al, 2002) or holons (Giret & Botti, 2004) which have been identified at a design stage. For the purposes of debugging it may be useful to view some agents as groups by virtue of their deployment on a physical network or others as groups because they form well established and trusted societies. Clusters may themselves contain other clusters so MAS can be considered as hierarchies of single-entity black-box forms which are expanded into collections of agents and other collections as the focus changes. Agents are only found at the lowest level of these hierarchies – their leaf nodes.

We have used this view of MAS structure as a basis for introducing rules of scoping and visibility. In this experiment, we allow portals to contain multiple agents and/or other portals. In addition we use the router meta-agent as introduced in last set of experimentation (a specialisation of portal which only attaches to other portals). Routers and portals know the location of any agent which they could legally forward messages to (ie: there is an entry in their routing table for that agent). The only constraint imposed on the possibilities for structuring virtual networks of portals and routers is that there should be no circular paths.

The notion of scoping is introduced in the follow way: agents declare their scope at registration, this scope can be described as "*global*" or can name a portal/router. *Global* agents are visible to all other agents, their details are included in the routing-tables of all portals and routers and any other agent in the system may send messages to a global agent.

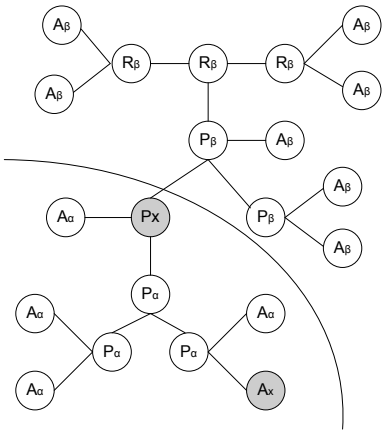

Figure 5: Visibility horizon for locally scoped agent A_x

When the scope of an agent is declared with the name of a portal/router, that agent is only visible to other agents in the same sub-MAS, ie: those agents which are located on the same side of the named portal/router. The details of the non-global agent are included only in the routing-tables of portals and routers in the same sub-MAS so only other agents in this sub-MAS may send messages to the non-global agent. Figure 5 shows the visibility of a non-global agent Ax declared with scope Px. Agents and portals Aα and Pα can locate and message Ax but those with β subscript cannot locate Ax.

Incorporating scoping in the manner described requires two modifications to the specifications of meta-agents. First, during agent registration, the details of any new agent are only propagated as far as the node (portal/router) named as its scope (with the obvious exception that if an agent's scope is declared as global then its details are propagated across the whole system). Secondly messages are not transmitted by a portal/router unless the recipient of the message has an entry in the routing-table of that portal/router (this is the normal behavior we have described but in some previous experiments we have allowed portals to transmit messages even when they were unable to locate recipients, in those experiments messages to unknown agents were transmitted to the nearest router which then attempted to resolve the location of named recipients).

In keeping with results of previous experiments, the addition of scoping was achieved with minor modification to the meta-agent subsystem. In this case modification was required only to the registration process managed by the portal-agent class. No specific change was required to routers since these inherit registration behavior from portal-agent.

3.4 Run-time Monitoring

Writing agent-based software imposes a level of complexity beyond that experienced with other approaches to software construction. One reason for this is that agent-based code is often more complex to debug than traditional software (Lam & Barber, 2004). This may be partly due to its reactive nature but additional problems are introduced by the distribution and concurrency of MAS where processing is spread across multiple agents and performance is as much a result of agent-agent interaction as it is a result of the data manipulations from individual blocks of program code. As with other programming paradigms run-time monitoring systems for MAS are important tools, there has been some discussion of ideas (Braubach et al, 2005; Lam & Barber, 2004; Lynch & Rajendran, 2008) and different MAS platforms provide varying levels of support (Bordini et al, 2006; Braubach et al, 2005).

Despite the need for run-time analysis of MAS, problems exist with trying to monitor a heterogeneous system of distributed agents who make independent decisions about their actions in autonomous ways. One central problem relates to the difficulty in collecting the necessary run-time information. A possible solution is to encourage/insist that programmers add code to their agents to capture their run-time status at specified points in execution and either display status information or relay it to some central monitoring system. However this is a poor solution, developers are unlikely to comply and, even if they do, would need to operate according to a set of standards for the nature and structure of status information which would impose additional burdens on development. Secondly, it is not clear how monitoring can be turned on/off as required without some specialised communication with each of the agents involved.

A better solution may be obtained by using meta-agents. Run-time monitoring systems of MAS can present various types of information. For the sake of the discussion here we concentrate on capturing information relating to the structure of a MAS and the messages exchanged between its application agents. Information about MAS structure can be obtained by examining the details of agent registration and messages exchanged between application agents are those sent between user-agents. Since portals route various meta-data, including that describing registration and user-agent messaging, portals can be readily modified to forward that meta-data to a monitoring system without disrupting any other system activity.

In practice we implement the monitoring system as a meta-agent (named "*monitor*" in the code below) and modify the message receiver of portals so the monitor is copied in to relevant information. There is a possibility that the monitor could receive duplicate or unnecessary data: as a message between user-agents is transmitted across multiple portals, each portal could forward the same data to the monitor. Similarly, as registration propagates to multiple portals they could report the same event (though in this case the data would not be the same because the mapping information changes as registration is forwarded between portals). To prevent the monitor from receiving duplicate/redundant information, meta-data concerning a particular agent is only sent from the portal which it is directly connected to.

To achieve these behavioral changes, the portal's add-agent method is extended in two ways (i) it forwards the name of both user-agent and the portal to the monitor (ii) it records the user-agent as an agent which is local to this portal (see use of is-local-agent below). The message receiver for the portal is then modified as follows:

```
method portal.msg-handler (msg)
| target =
|  routing-table.get(recipient-of(msg))
| if is-local-agent(target) then
|    target.enqueue( msg ))
|    routing-table.get(monitor)
|                      .enqueue( msg )
| else socket-agent.enqueue( msg ))
```

This approach has allowed us to successfully incorporate a run-time monitor into any MAS built using meta-agents.

4 CONCLUSIONS

This paper has highlighted a limitation with the agent platforms and middleware which are currently available – they are not designed to allow system developers to modify their behaviour so cannot be tailored to suit the needs of developers. By implication, notions of agency, the characteristics of operation and the types of agent-agent collaborations that can occur are not determined wholly by MAS developers but also by the platforms they use. Additionally systems architectures (whether hub-based or peer-to-peer) and the nature of inter-agent and cross-network communication channels may be fixed. This imposes a significant constraint on developers who must partially shape their systems to fit the restrictions of their chosen platform rather

than the needs of their applications or deployment circumstances.

These types of limitation not only affect MAS platforms but are typical for many other software systems, notably programming languages which traditionally have had fixed semantics that cannot be tuned by application programmers. Some programming languages have recently attempted to overcome this restriction using metaobject protocols which provide a programmer interface into the language so that its behaviour may be altered by programmers. Influenced by the design goals of the metaobject protocol and related work on meta-agents and actors we have specified a small set of meta-agents, light-weight components which lend themselves to modification and may readily be configured into different patterns.

We have shown how some patterns of interacting meta-agents can be made to form distributed subsystems which function as platforms and middleware for higher-level application agents and have shown how different configurations of meta-agent subsystems exhibit different properties and how these properties impact on the characteristics of applications using them as their middleware. In addition, we have implemented meta-agent subsystems in various programming languages and linked them to other, third party, systems.

In a series of experiments we have modified and reconfigured meta-agent design patterns with predictable effects on the middleware of systems built from these patterns. In this way we have demonstrated that it is possible to use meta-agents to build extensible MAS platforms which may be modified in order to suit different MAS applications and their requirements for deployment.

REFERENCES

Bellifemine, F. L., Caire, G., and Greenwood, D. 2007. Developing Multi-Agent Systems with JADE, ISBN: 978-0-470-05747-6. Wiley.

Bellifemine, F., Caire, G., Poggi, A., and Rimassa, G. 2008. JADE: A software framework for developing multi-agent applications. Lessons learned. Inf. Softw. Technol. 50, 1-2 (Jan. 2008), 10-21.

Bordini, R., Braubach, L., Dastani, M., Seghrouchni, A.E.F., Gomez-Sanz, J.J., Leite, J., O'Hare, G., Pokahr, A., and Ricci, A. “A Survey of Programming Languages and Platforms for Multi-Agent Systems”. Informatica, 2006, 30(1), 33-44.

Braubach, L., Pokahr, A., Bade, D., Krempels, K., and Lamersdorf, W. 2005. Deployment of Distributed Multi-Agent Systems. M.-P. Gleizes, A.Omicini, F.Zambonelli (Eds.): ESAW 2004, Springer-Verlag, Berlin Heidelberg, LNAI 3451, 261-276.

Cabri, G., Ferrari, L., Leonardi, L. and Quitadamo, R. 2006. Strong Agent Mobility for Aglets Based on the IBM JikesRVM. ACM symposium on Applied computing (Dijon, France) ACM.

Cheyer, A. and Martin, D. “The Open Agent Architecture”. Journal of Autonomous Agents and Multi-Agent Systems, 2001, 4, 143-148.

Dastani, M. 2008. 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems 16, 3 (Jun. 2008), 214-248.

Fedoruk, A. and Deters, R. 2002. Improving Fault-tolerance by Replicating Agents. In Proceedings of AAMAS: part 2. (Bologna, Italy, 2002) ACM Press, New York, NY.

Fonseca, S. P. 2006. Engineering degrees of agency. In Proceedings of the international workshop on Software engineering for large-scale multi-agent systems, SELMAS (Shanghai, China, 2006) ACM Press, New York, NY.

Giret, A. and Botti, V., “Holons and Agents”, Journal of Intelligent Manufacturing 2004, Vol. 15 No.5 pp. 645-659. Springer Netherlands.

Gutknecht, O. and Ferber, J. The Mad Kit Agent Platform Architecture. In Infrastructure for Agents, Multi-agent Systems, and Scalable Multi-agent Systems, 3-7, (2000)

Kiczales, G., Rivières, J.D., and Bobrow, G.D. 1991. The Art of the Metaobject Protocol, MIT press.

Kind, A. and Padget, J. Towards Meta-Agent Protocols, LNCS 1624, 30-42, 1999

Lam, D. N. and Barber, K. S. 2004. Verifying and Explaining Agent Behaviour in an Implemented Agent System. In Proceedings of AAMAS (New York, USA, 2004) ACM Press, New York, NY.

Lynch, S.C. and Rajendran, K. 2008. Providing Integrated Development Environments for Multiagent Systems, Multiagent System Technologies, Springer-Verlag, Berlin Heidelberg, LNAI 5244, 123-134,.

Mascardi, V., Demergasso, D. and Ancona, D. 2005. Languages for Programming BDI-style Agents: an Overview. In Proceedings of WOA (Camerino, Italy, 2005). Pitagora Editrice Bologna.

Massonet, P., Deville, Y., and Neve, C. 2002. From AOSE Methodology to Agent Implementation. In Proceedings of AAMAS (Bologna, Italy, 2002) ACM Press, New York, NY.

Mulet, L, Such, J M. and Alberola, J M., 2006, Performance evaluation of open-source multiagent platforms. AAMAS (Hakodate, Japan) ACM Press, New York, NY

Sen, K., Agha, G., "Thin Middleware for Ubiquitous Computing," Process Coordination and Ubiquitous Computing, CRC Press, 2002.

Suna, A. and Fallah-Seghrouchni, A., E. 2005. A Mobile Agents Platform: Architecture, Mobility and Security Elements. LNCS, 3346 / 2005, 126-146.