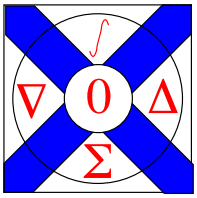

Computational Science Technical Note **CSTN-054**

Software Integration Architectures for Agents

K. A. Hawick and A. P. Gerdelan

2008

Many modern artificial intelligence (AI) systems, including both real physical robots and animat-based models are structured using intelligent agents. A key challenge for building large and complex AI systems is to manage the agent interactions in an appropriate architecture that supports complexity in a scalable and hierarchical manner. We review various agent architectures for both physical and simulated robot systems, and show how appropriate agent communications protocols can be developed to support artificially intelligent systems based on communities of interacting agents. As well as reviewing some of the architectures and supporting software tools and technologies, we present our own ideas for a software architecture for managing intelligent agents. We emphasise the importance of being able to incrementally augment the set of agents as new ideas are developed. We describe how key activities such as agent navigation in physical and simulated spaces; agent communication; world state management and sensory integration all need to be managed in an appropriate framework to support individual agents that will take responsibility for tasks and goals.

Keywords: AI; agents; software; architectures; path algorithms

BiBTeX reference:

```
@TECHREPORT{CSTN-054,
        author = {K. A. Hawick and A. P. Gerdelan},
        title = {Software Integration Architectures for Agents},
        institution = {Computer Science, Massey University},
        year = {2008},
        number = {CSTN-054},
        address = {Albany, North Shore 102-904, Auckland, New Zealand},
        month = {May},
        keywords = {AI; agents; software; architectures; path algorithms},
        owner = {kahawick},
        timestamp = {2012.12.28},
        url = {http://www.massey.ac.nz/~kahawick/cstn/054/cstn-054.pdf}
}
```


Prof Ken Hawick, Computer Science, Massey University, Albany, North Shore 102-904, Auckland, New Zealand.

Technical Report CSTN-054

Software Integration Architectures for Agents

K.A. Hawick and A.P. Gerdelan
Institute of Information and Mathematical Sciences, Massey University
Albany, North Shore 102-904, Auckland, New Zealand
Email: k.a.hawick@massey.ac.nz, a.gerdelan@massey.ac.nz
Tel: +64 9 414 0800 Fax: +64 9 441 8181

May 2008

Abstract

Many modern artificial intelligence (AI) systems, including both real physical robots and animat-based models are structured using intelligent agents. A key challenge for building large and complex AI systems is to manage the agent interactions in an appropriate architecture that supports complexity in a scalable and hierarchical manner. In this article we review various agent architectures for both physical and simulated robot systems, and show how appropriate agent communications protocols can be developed to support artificially intelligent systems based on communities of interacting agents. As well as reviewing some of the architectures and supporting software tools and technologies, we present our own ideas for a software architecture for managing intelligent agents. We emphasise the importance of being able to incrementally augment the set of agents as new ideas are developed. We describe how key activities such as agent navigation in physical and simulated spaces; agent communication; world state management and sensory integration all need to be managed in an appropriate framework to support individual agents that will take responsibility for tasks and goals.

Keywords: AI; agents; software; architectures; path algorithms.

1 Introduction

Artificial Intelligence techniques have found their way into many complex software systems including game-engines, vehicle [1] and robot control systems [2]. Many artificially semi-intelligent agents are used in everything from experimental smart cars to artificial life simulations. Various cut-down and built up incarnations of what are essentially agents based on the same core principles but with different attachments are made to represent everything from computer generated "Orukai extras" in Lord of the Rings movies to Soviet tanks in the NZ Army's battlefield simulation. There are both agents that are made to squeeze into a small window of CPU cycles and just need to make an animated character look convincing, and agents that can use a bit more computational power and genuinely need to be a bit smarter. There are various run-again-until-it-doesn't-do-something-dumb agents for cinematic movie usage and those that need to run in real time for games. There are different architectural tradeoffs in these requirements, but some common features too.

Many software control and robot control systems have evolved incrementally – often built from the lowest level of hardware drivers upwards, and it is therefore useful to reflect on how these systems might be re-engineered using a more completely thought-through architecture. In principle an agent-oriented approach may help achieve this goal.

Figure 1 shows a typical example. A "swarm" or collective of robots such as the group of Cybots [3] shown, must collaborate on a task such as finding an object. Some five years ago it would have been necessary to locate the control software entirely on a separate base station and to control individual robots or agents remotely. This is still possible and a good approach for some problems. It is now becoming feasible for each separate device to have embedded in it a quite powerful and semi-autonomous computer that runs its control program and hosts its "artificial intelligence." However, even in this case from an architectural per-

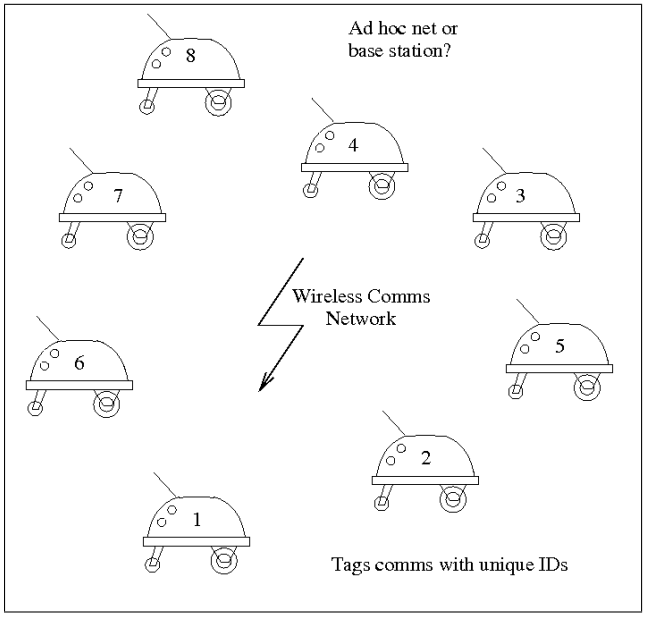

Figure 1: A swarm community of agent controlled robots that may have autonomous on-board computers or may be managed from a base-station.

spective we must still think of the system as a whole - as a set of software agents that must communicate and cooperate amongst one another to achieve the overall goal. Where the agents are hosted is from an algorithmic perspective immaterial. There are of course still many software architectural practicalities that do dictate the performance of a solution however.

There are already a number of software architectural approaches and models that can be used for multi-agent [4] control systems [5, 6]. It is not yet obvious that any particular one is best, although we are somewhat drawn to a middleware approach. The key problem seems to be managing interactions and software complexity. These software frameworks are of necessity quite large and it is important to find a way in which some separation of function and modularity can be introduced. The problems are similar to those encountered in distributed and remote object models. Historical systems have exposed access to what should be hidden low-level driver details to high software functions that have thus become too closely coupled to low-level details. This was perhaps necessary in the past to obtain workable performance from the systems. At present however it is likely that progress is being held up by software complexity and an inability to introduce new algorithms due to many agent control software systems being too disorganised to handle new features.

Agents can be programmed in a number of languages. Some of the Artificial Intelligence languages such as the Lisp [7] family of languages or Prolog [8, 9] families lend themselves well to simulations and to elegant reasoning structures [10–13] but are not necessarily capable of real-time performance and control of hardware devices. Likewise in the case of very many animat [14] agents in a simulation, languages like C or C++ are still necessary [15].

In this article we discuss software architectural ideas for agent-oriented systems in particular. These ideas have been employed for animat agents in complex system simulations [16, 17] as well as for software agents in computer games and for controlling real physical robots.

This article is structured as follows. In section 2 we describe some important uses concerning inter agent communication and present some ideas for implementing agent and robot communication protocols in section 3. We review ideas on software architectures in section 4 focusing particularly on middleware ideas in section 5. We give a discussion of agent environment modelling, behaviour, decision making, coordination and redundancy in section 6. Finally in section 7 we give a summary of our present thinking and some future directions for development.

2 Inter-Agent Communication

Inter-agent communication is not a trivial problem. Its potential benefits would make communication more localised and relevant in the same way as peer-to-peer Internets use. More robust networks of agents ask neighbors for help rather than wait for orders from single controller. It is not effective in some scenarios - e.g. simulated agents probably use more CPU (because they don't have dedicated compute resources) in multi-communication than in master-slave communication.

It has been our experience that because of the wide variety of sensors and interconnection device types on the market, there is definite need for a simplified method for control. What is needed is a generic GUI and middleware software layer that allows high-level instructions and conditions to be passed to a sensor or a group of sensors so they could be programmed as a cohort. We report on work-in-progress.

The method of device update through the wireless link is of importance. The basic decisions are either a push model or a pull model, with varying degrees between each. At one extreme, the push model stipulates that the main server sends out all information to the sen-

sors and in this process they are essentially passive; it also requires a reliable broadcast (multicast or unicast) mechanism. At the other extreme a pull model puts the responsibility for retrieving updates on the sensor; the sensor must poll the server for updates at regular intervals, which may waste precious bandwidth.

We favour the event-notification approach taken by the designers of the Java language system. In this model sensors that are interested in particular events or conditions register themselves with a server or group of servers. The same happens in the reverse direction: servers are able to register themselves as being interested in an event or condition that may arise from the sensor. When the specified event occurs, whichever sensors or servers are registered as listeners for that event are notified in a push-type model. This approach keeps events high level and therefore makes software cheaper to develop and maintain. It is also possible to construct priorities with event classes so that more important information is able to be acted upon by servers, etc more rapidly.

We have implemented our middleware system using Java as this language system provides GUI components, access to the network, distributed computing facilities and serial interfaces that are necessary to achieve our aims. We have recently also begun experimenting with other technologies such as Jini and JavaSpaces as a high-level communications substrate. We have had recent success in porting the Jini and JavaSpaces environments to run on the TINI boards, which only have 1Mb of RAM; this process has necessitated writing space efficient code using socket based communications. However, we expect the memory available on such devices to become a) more plentiful and b) cheap rapidly.

3 Communication Protocols

We are experimenting with simple protocols based on a broadcast of a vector or list of device tags. This can be readily implemented using a message passing broadcast using Java based middleware we developed for embedded parallel programs [18, 19]. One of the most simplistic protocols involves the exchange of server lists; as the devices learn about more servers, the list becomes larger until the device has a complete (as possible) knowledge of the system state. This process is repeated at varying intervals depending on the dynamicism of the environment

There are obvious scalability limits to the simplistic protocol we are using at present. Nevertheless this is still adequate for a few hundred devices updating relatively slowly. Our system relies on a unique a well known name space for the devices. It is an interesting problem to consider dynamic discovery protocols [20] where devices are essentially unknown and untrusted. This is the resource discovery problem faced on global grid networks.

Once the self-configuring sensor or robot net has established a route from each node to each other node a variety of interesting sensor update or querying operations are possible. Updates in the presence of partial or temporary disconnection can be tolerated by assuming an asynchronous or non-blocking communications structure and by using a time-stamping protocol to propagate delayed readings around the system. Our assumption is that at any time there may be some temporary disruption or delay to messages but that the nature of the application allows this to be tolerated.

Recent technological and economic advances in wireless devices make it feasible to deploy *ad hoc* networks of sensors and robots. As the costs of the electronics drops, the cost of deployment is increasingly dominated by the cost of control and interoperability software.

We are working to develop a software control fabric based on a Java middleware. We have used WaveLAN technologies to prototype our system and are actively working on Bluetooth interoperability for deploying groups of cooperating mobile robots. Our Bluetooth efforts are still in the prototype stages; we are investigating access to the Bluetooth network stack from a high-level language such as Java.

There has recently been considerable interest in wireless Internet-connected devices such as Personal Digital Assistants(PDAs) and mobile phones. Recent industry standards for wireless communications, such as Bluetooth [21] and IEEE 802.11b [22] are allowing greater degrees of flexibility and interoperability between mobile devices. The commodity pricing of wireless kits makes it economically feasible to consider wireless networks as an alternative to installing fixed networks using cables for some relatively low bandwidth applications such as sensor networks and other monitoring devices.

A related project involves the use of wireless communications to allow a "swarm" of mobile robots to inter communicate and work on shared tasks. We are building robots controlled by the TINI [23] processors from Dallas Semiconductors and are presently work-

ing to interface these with Bluetooth radio devices. Our development environment makes use of WaveLAN 802.11b wireless Ethernet cards inter-operating with (relatively) small mobile computers such as the Compaq iPAQ [24].

One of the hardest tasks of modern computing is keeping track of the myriad of devices that are connected to a network. This problem is magnified if the network allows sporadic mobile connections. One can foresee almost insurmountable problems when, if predictions come true, every device in our lives including our refrigerators, have their own IPv6 address and are connected to the Internet.

Possibly the most significant factors that constructing a sensor net brings to light are those of reliability, on-board compute power, cost and the range of such device. As the sensors are constructed from COTS hardware, and are all self-contained, then the reliability of individual devices is a concern; especially if we run a proper sensor net, in which messages are being relayed from sensor to sensor until they reach their targets, it would be catastrophic if components were to fail thus segmenting parts of the network. One trivial remedy for this problem is to ensure that sensors are placed sufficiently close together that they have signal overlap with at least two others. One of the methods we are researching in order to detect failure of any device is to send a 'heartbeat' or 'systolic pulse' through the system. This topic will be explored further when we discuss our software architecture in section 4.

The cost of the device and its relative on-board power are closely related. While a full PC processor would provide a large amount of on-board processing power, at present it is not economic and if this approach were taken it would only be for very specialised sensors requiring a lot of pre-processing of acquired data. We have also experimented with sensors that have an embedded Compaq iPAQ [24]. This device runs the Pocket PC operating system and have interfaces to serial ports, that can connect to sensor inputs and outputs; as with the PC processors, cost still makes it prohibitive to equip all sensors with these devices. As explained earlier in this paper, we are concentrating our efforts on the provision of our sensors with the simpler TINI boards, each having 1Mb of RAM and a native Java engine on-board. Coupled with a suitable networking technology such as 802.11b wireless Ethernet or Bluetooth, the cost per sensor drops to a more economic amount.

The range of Bluetooth can vary from 10m to 100m depending on what power radio transmitter is being used. Of course, power consumption is proportional

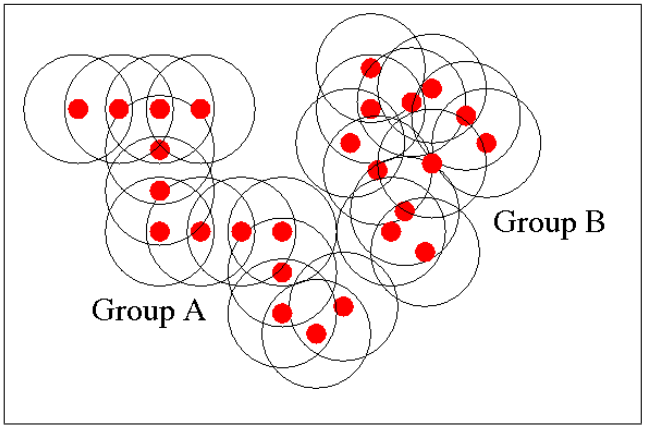

Figure 2: Sensors equipped with a limited range wireless communications system. Circles indicate the physical range of communication of each unit. Group A is laid out as might be used in corridor or door sensors in a building. Group B is a more random layout. The groups are distinct as there is no unit in place to span them.

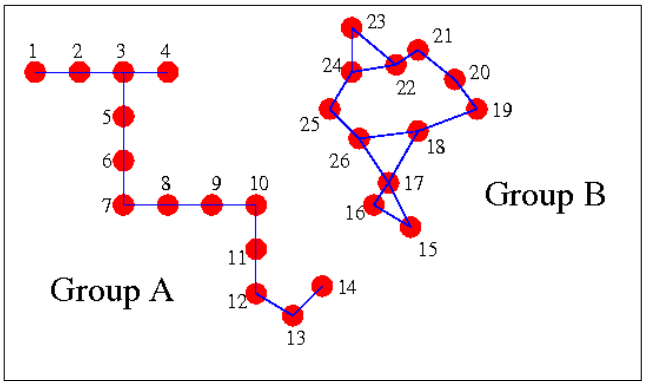

Figure 3: A graph model with identified adjacencies shows that robots 1-14 are part of group A whereas robots (or agents) 15-26 are part of group B. This also governs which neighbour will be used to pass on communications for a particular targeted destination robot.

to the output power, so it is in our best interests to tune this to keep the output as low as possible. We are currently investigating various tuning mechanisms.

The system can be modelled as shown in figure 2. A number of sensor devices each with a range are located so as to overlap the ranges – drawn as circles. The individuals are able to resolve that they belong in two separate groups of graph clusters - as shown in figure 3.

4 Architectures

Software architectures can be structured and modelled in a number of ways including as a simple stack or as a middleware infrastructure.

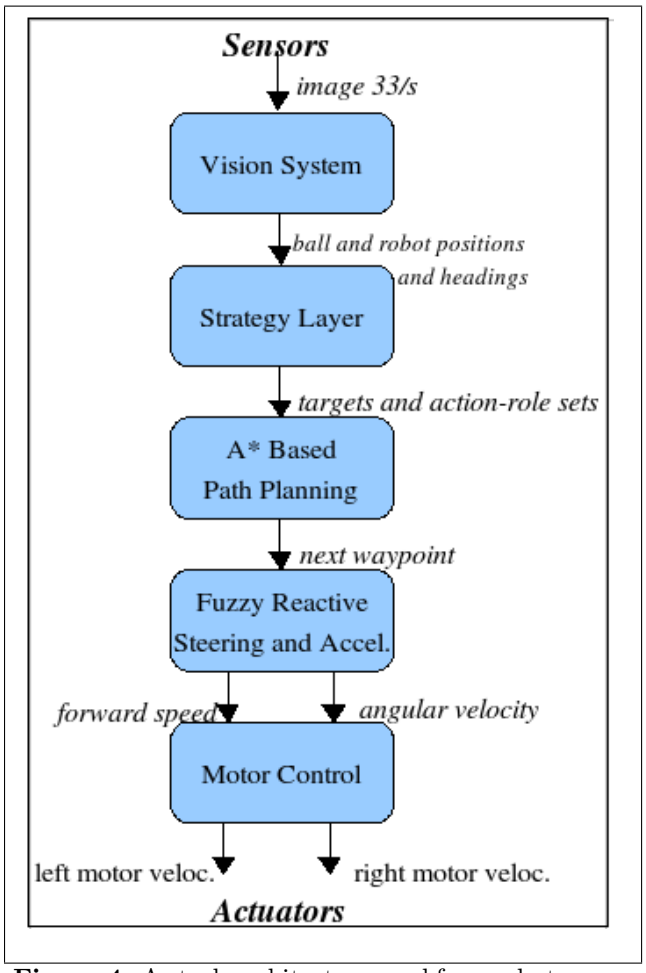

Figure 4: A stack architecture used for a robot soccer player agent.

Figure 4 illustrates a typical stack-based intelligent agent. In this case the agent is a controller for a robot soccer player [25]. Whilst the vision system that processes information from the sensors, and motor control layer for communicating with two small engines are unique to the agent, the decision-making and path-planning layers in-between are not. Indeed, we have shown that an identical layered stack architecture can be used for simulated agents with different sensory and actuator configurations, and in a variety of different environments in both 2 and 3 dimensions [26] [27] [28] and is equally relevant to both simulated and physical agents.

Middleware is a software layer that sits between top-level applications and the operating system in distributed computing systems. Although the use of intelligent agents has expanded to operation in a huge variety of environments, many of the difficult but interesting problems of agent behaviour remain the same, and it becomes convenient to locate the programmed model in a smart middleware.

The typical stack architecture of robot agents, such as that illustrated in Figure 4 lends itself amenable to the middleware approach; the sensor controller and actuator controller layers of the stack must be designed separately, but under a middleware-based agent, would then only have to provide interfaces to a middleware layer capable of handling the path planning, goal prioritising, and decision making behaviours of the agent.

Whilst there are an increasing number of highly specialised jobs or scenarios for which cooperating intelligent agents are being designed, emerging from these specialised applications we observe that that the processes and core software architecture required by these agents is often very similar. Indeed we find from our own work, even from independently conceived systems, that we have often repeated nearly identically the foundation work behind agents developed for a variety of different environments; both simulated and robotic [29] [26] [27] [28].

Our experiments with agents in Artificial Life, battlefield simulations, robot soccer, and other simulated and robotic systems all make use of agents; the requirements of which we observe converge towards a similar model; large numbers of relatively simple agents that operate autonomously within their environment. More complex functionality is then either emergent (as in the case of Artificial Life) or is directed by higher-level agents or humans (as in the case of battlefield simulations and robot soccer). Some of these low-level agents are intended to cooperate in a swarm arrangement, or compete with hostile or predatory agents, but all have a number of elements in common:

A short-sighted perception of the surrounding environment

An evaluation system to decide which action to take next

Navigation functionality (a system that decides where to move)

An ability to record a simple history of past actions

A framework for interacting with other agents

Other types of agents such as natural language processors may have a radically different *modus operandi*, and require a specialised architecture, but for systems that employ agents in an autonomous or mobile context we propose a new architecture; with an agent middleware to handle integration or and communication between the common agent elements as listed above. This middleware is to be constructed in a modular fashion; where the elements of agent functionality can be swapped-in or swapped-out, redefined, or extended to better suit the agent. This architectural approach should then be flexible enough to cope with a variety of heterogeneous agents within a common framework, and as we will show, even lends itself supportive of more complex or distributed agents.

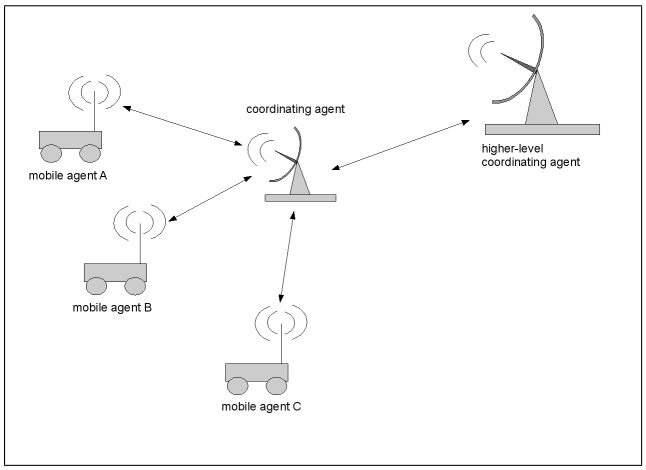

Figure 5: A society of agents based on a hierarchical organisation.

Figure 5 illustrates the model for inter-agent cooperation and communication that we have had in mind when designing this agent architecture. Whilst recent research has looked at the possibility of peer-to-peer type communication models for agents [30], we have found from experiments with our own agents that a society of cooperating agents based around a military-style hierarchy (or *team captain* in the case of Robot Soccer) is the most effective for practical applications. The autonomous agents are relatively non-complex, with mostly reactive behavioural intelligence capability, but act as eyes and ears for a coordinating agent to which they send asynchronous reports. This coordinating or higher-level agent processes environment information reported from its agents, and from a more complex model of the environment makes more complex forward-thinking plans relating to a common goal, and then sends *missions* to the agents under its umbrella in order to achieve best results through cooperative actions; goals which otherwise might not serve the highest interests of the individual agents. Because many multi-agent systems are constrained to an individual machine with limited cycles available for inter-agent communication and cooperation, the hierarchical model of communication also makes much more efficient use of resources for these systems.

5 Modular Middleware

Our concept for the role of an agent middleware for our agents is illustrated in Figure 6. We have identified some of the key tasks common to our agents, which we have included as tasks that will be handled by our agent middleware. We also find that each agent we create will always require agent-specific modules for interfacing with the unique actuators and sensors of that agent; be they software or hardware. It is however, possible for these calibration and control modules to be separated from the core of the agent, and communicate to an agent middleware via a common interface and communication protocol. It is not necessary to build a completely new agent architecture from scratch simply because one agent must operate physical motors as actuators, and another that employs the same algorithms exists only in a simulated environment.

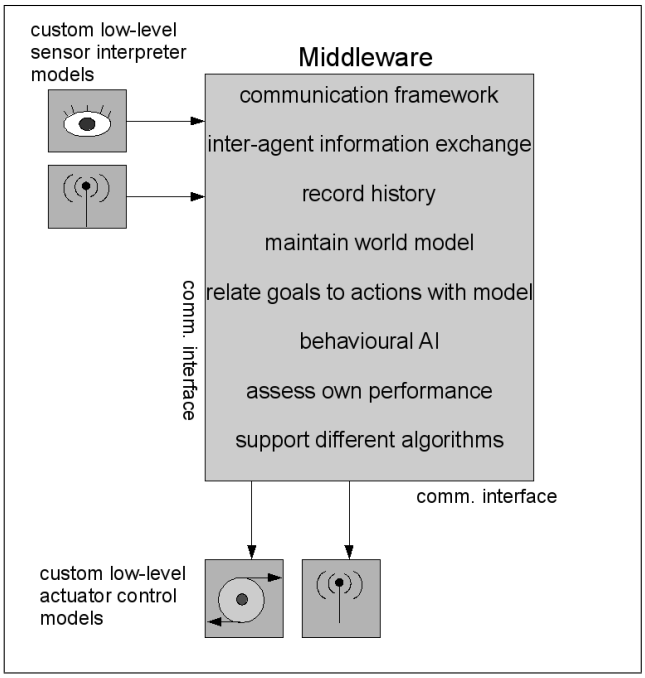

Figure 6: The role of middleware for our agents.

A progression of the agent middleware concept in our architecture is a move to modular middleware. Separating the functional components of our agent middleware into modules, as illustrated in Figure 7 allows

us a much greater degree of flexibility in design, which is particularly useful if we want to develop an agent middleware that can be applied to various types of agent with differing functional requirements.

Within *main-loop* style agent control programmes the tradition has been for all the different functional components of an agent to be computed once per frame. This model is not an efficient use of resources, as some of the typical agent modules such as long-term path planning do not need to be re-computed in every processing frame. If the modules of our agent middleware are created in an asynchronous fashion then they can re-calculate on independent terms, and simply respond to the requests from other modules with the whatever the latest data is. An asynchronous module design with common interfaces allows us to spread calculation load over many processing cycles.

A modular architecture also allows the designer of an agent to replace any of the modules with some of the designer's own code, or with a module from another agent. With an object-oriented implementation the designer should even be able to extend or override the core functionality a module to add some more specialised functionality to the agent.

This semi-independent modular approach also gives us the freedom to distribute our agent middleware over several threads or multiple CPU cores, or by logical extension to distribute our middleware for a single agent over geographically separated machines as agents become more resource-hungry or require greater internal redundancy.

The architecture that we have designed is **scalable** in two senses; firstly, the modular nature of the agents allows upgrade or downgrade in complexity. Because the modules themselves can be constructed to communicate asynchronously, through a common interface, this architecture allows entirely new modules to be included within the architecture. An example of where this approach is most useful is the navigation stack of autonomous agents; which can typically comprise several different systems for long distance map-based planning, short distance path-finding, reactive obstacle avoidance and target seeking, and several levels of interface with actuators. For an example see Figure 4. The number of systems in the stack is largely up to the designer of the system, and it is definitely beneficial to offer this kind of flexibility to designers of agents, or to allow agents with different levels of navigation complexity to operate under the same architecture.

Secondly, the society of these agents itself is scalable; if there are large numbers of agents then coordinating agents can be stacked in a hierarchy - coordinating agents control a group of agents as actuators, and report to a higher-level coordinator as its sensors (see Figure 5). Each level of coordinating agent deals with a model of the environment at a different resolution and sends and receives information at a different level of abstraction. Each agents knows only about the agents directly above and below it in the hierarchy. This model minimises conflict between cooperating agents as it scales in size as agents at each level can be given missions that avoid competition (or collision) by the level above.

6 Discussion of Design Issues

There are a number of design features and the associated issues to consider and which we discuss in this section. These include: support for an environmental model; agent behaviour models; agent decision making; agent coordination; and provision for redundancy. These are of course related, and have important implications for am architectural design.

6.1 Modelling the Environment

A very simple method for modelling most agent environments also has a military-style planning analogy; in the same way as infantry radio operators give simple grid-reference coordinates for artillery or airstrikes, we will use a similar model for some of the same reasons. The perceived world is plotted on a 2-dimensional map which is divided into map references or cells. All elements of the environment in the agent's word model are then said to occupy a discreet map reference, rather than an in-exact position expressed in floating point or real numbers. There are several advantages to this approach:

The world model can be stored in 2d arrays.

Reduced error when communicating positions.

Ease of computation for common algorithms.

Ease of pattern identification.

A 2d map is a very simple tool for communication - it is usually not necessary for agents to communicate very accurate world information; this sort of information is usually only required for very low-level planning. Using integers to express map cells removes any

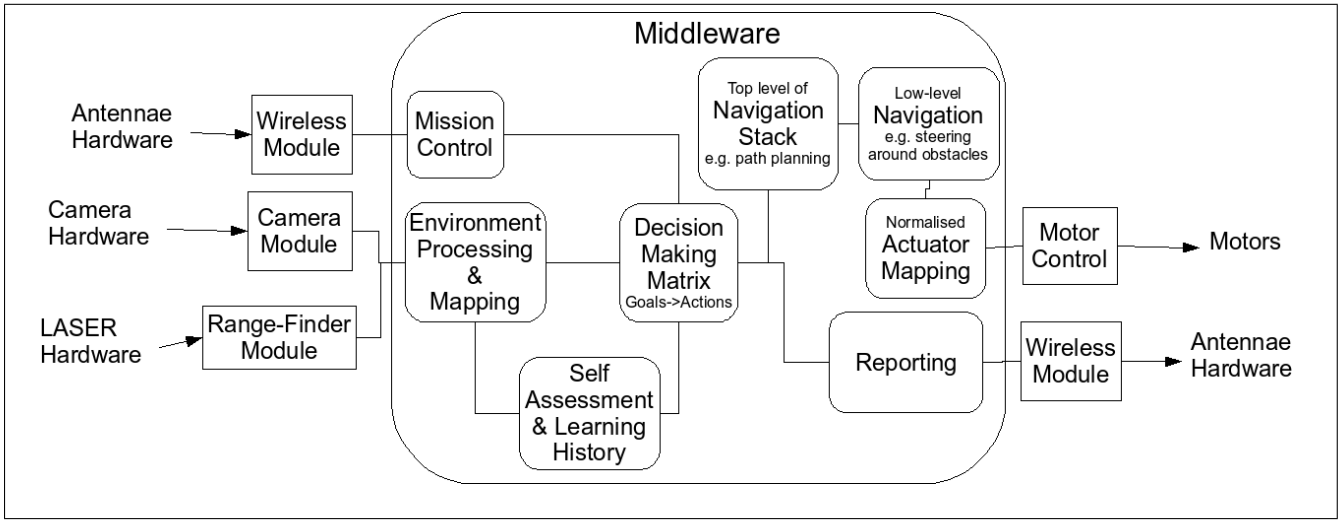

Figure 7: A modular middleware architecture with exchangeable components.

uncertainty that an agent based on a different machine architecture will interpret the data differently, and reduce the error in communication.

The resolution of the map is obviously important; lower resolution maps are easier for planning, but are at risk of losing important or subtle data. Maps of higher resolution contain more information but are at risk of bogging down algorithms with unnecessary information, over-sensitive grid positions for moving objects and exponentially expanded search domains. Our previous experiments have shown that an ideal map cell size was roughly 150% of agent size for Robot Soccer agents requiring very accurate navigation [26].

A key advantage of using a world model as simple as this is that the thoroughly tested and proven algorithms of computer science can be applied to good effect. The A* search algorithm or other real-time adaptations and hybrids can be used for very effective path planning. Common pattern identification and image recognition algorithms can be applied to the map for use in higher-level artificial intelligence. At any time the map can be quickly transmitted as a human-recognisable image. Lossless compression algorithms can even be applied to the world models in the same way as they would an image, for more efficient transfer of very large or detailed world maps.

Figure 8, which we shall explain in detail shortly, shows the similarity between this sort of world model and the famous Wumpus [29] problem. What we are really trying to achieve is to take what may be very complex 3D world information from the agent's sensor modules and convert this into a very simple view of the world - in much the same way as humans would - we simplify the very complex problem, then solve

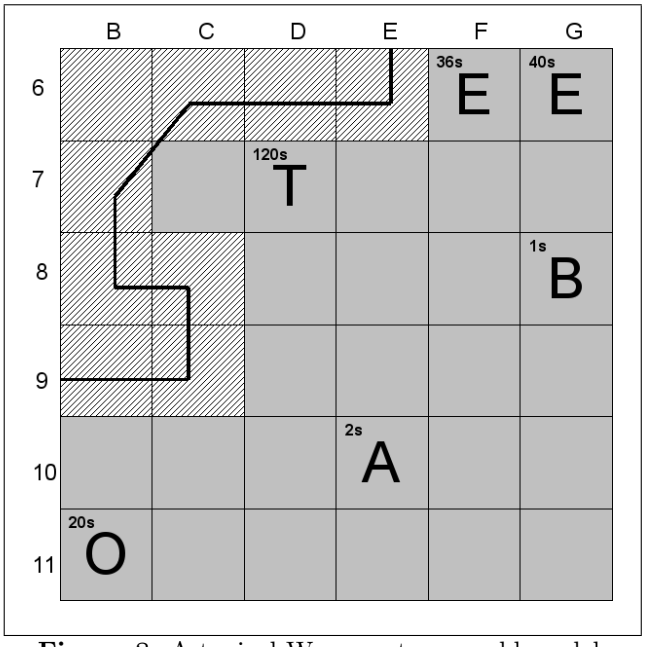

Figure 8: A typical Wumpus-type world model.

the simple problem and apply it back to the complex domain. We plan using this simplified and abstract model of the world, making high level plans such as "from our position move around this map cell to get to destination map cell" which are then interpreted by an intermediate control module into low-level complex outputs which can be fed to actuator control modules.

The main requirement in order to harvest this sort of high-level planning ability is a system for converting to and from complex and map coordinates. This system depends on the type of environment that the agent is operating in; systems have already been developed for

robots operating in the physical world for exactly this sort of mapping and planning. Carnegie Mellon University's Lunar Rover is a very good example of this sort of system; and provides details for a stereo-vision camera system to identify positional information of terrain features, and plot them on a similar type of map for later use as a navigational aid [31]. Simulated agents may already operate in this sort of cell-based environment, but those that operate in a more complex 3d space simply require a system for agreeing on map cell placement, but systems for this purpose have also been developed [26].

In reference to Figure 8, we can consider the agent marked 'A' at map index (E, 10). The superscript in the top left corner of the cell indicates that this information was last updated 2 seconds ago, which shows us that we can actually store several layers of useful information in each cell, and also indicates that this sort of planning is useful in a dynamic and uncertain or stochastic environment; we can *datestamp* world model information which can assist high-level planning modules in the assessment of certainty or out-of-date information. This type of information is useful in identifying a pattern of nearby hostile agents as being actually only one or two moving agents that have been reported multiple times, or indicating if a segment of the environment containing dynamic obstacles will no longer be reliable for path planning. Again in the figure, we can see that that our agent has been informed of, or spotted first-hand, various other environment elements with their own datestamp. Each of these elements has been categorised with a different letter; this kind of very simple discrimination is useful for fast-paced decision making such as real-time path planning, assessing danger levels, or roughly categorising the possible speed of dynamic obstacles. A large segment of the map is also shaded out completely - a wall has been discovered and all of the cells that it covers or mostly covers have been removed from the navigation search domain. This is a blanket decision, but will reduce the load on long-range path finding. Lower-level navigation behaviour may actually move the agent through the parts of these map cells that are not impeded if necessary. We can see that in this way we have drastically reduced the complexity of our world into a model not dissimilar from the world of the Wumpus, and for good reason - the powerful algorithms that have been developed for Wumpus-like problems can then be applied to complex domains, or even the real world.

6.2 Agent Behaviour

Within this architecture agent behaviour must be decomposed at several levels of abstraction. For this reason we typically have *navigation stack* type sub-architectures which sit in-between the decision-making module and each of the agent's actuators. This concept is illustrated in Figure 4. Some actuators such as wireless transmitters may not require a large stack, but simply a module to encode a message into a format appropriate for the communication protocol and pass that to the wireless adapter. Other actuators, particularly those used for complex problems like 3D navigation typically require a string of modules, each processing a different level of the problem space; filtering higher-level instructions down until specific actuator instructions are computed.

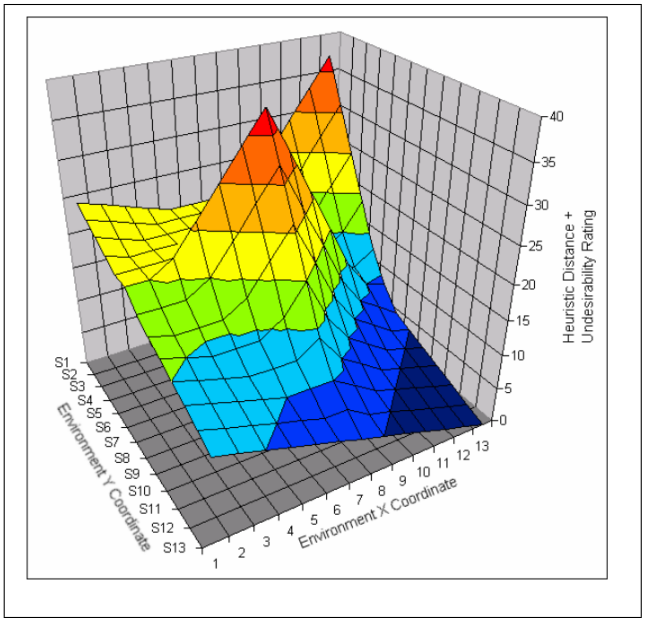

Figure 9: Heuristic representation of the FuzzyA* navigation algorithm.

An agent navigation system within this architecture typically starts operation on the agent's 2d world map. High-level algorithms such as A* or modifications of A* like Hybrid Evasive Fuzzy A* [27] or other search algorithms can compute useful long-distance or mid-range real-time planning information here, based combined heuristics as depicted in Figure 9 assembled from various layers of the Wumpus-style environment maps. The Fuzzy A* algorithm can go so far as to generate forward-thinking probabilistic and regions of *undesirability* and factor these into a balanced path-finding algorithm, so that mobile agents can take a shortest path, excepting areas that *are quite likely to be undesirable*, for example a tear-drop or wave-crest

shaped region surrounding an approaching heavy vehicle. An example of this algorithm in simulation is illustrated in Figure 10.

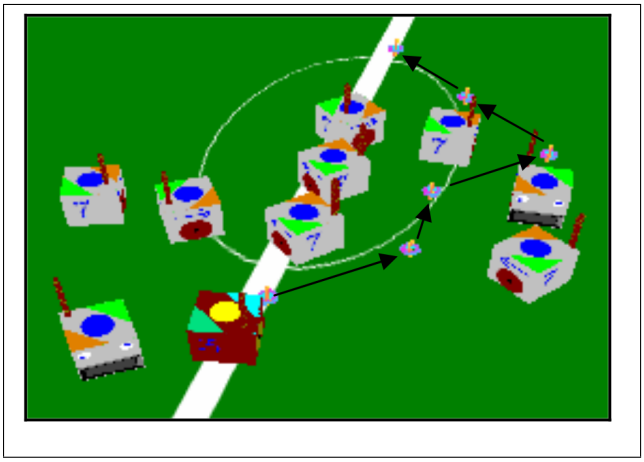

Figure 10: Real-time dynamic path-finding in a robot soccer simulation.

Because effective navigation systems generally operate dynamically in real-time, generally only the *next* waypoint in the planned path is fed down from the planning layer to the next layer in a navigation stack. That next layer generally takes care of more subtle details, but generally attempts to direct the agent in a straight line towards the next waypoint in a *target seeking* approach. In the case of the Fuzzy A* algorithm, this next layer is more complex; and actually computes a Fuzzy Logic-based reactive obstacle avoidance steering and acceleration system combined with a Fuzzy Logic-based target seeking system. This kind of control gives the agent a reactive navigation layer, and can quite sensibly control analogue mechanical actuator systems.

Subsequent modules in the navigation stack would then be required to translate the generalised defuzzified steering and acceleration outputs into specific motor instructions, but here a module would need to be developed outside the middleware to convert outputs into actuator controls based on formulae specific to the nature of the agent.

6.3 Decision Making

The key function of an intelligent agent is the decision-making process; or how the agent maps its goals to actions. Non-complex agents such as Animats [29] are typically provided with a list of possible actions, and either a procedural or heuristic method for ranking these actions in order of priority based on information held about the current state of the environment, and any other information held by the agent. This model should be acceptable for simple agents, such as the mobile or bottom-level agents in the society.

Higher-level coordinating agents deal with a large database of different types of information and can manage a large number of actuators - their subordinate agents - simultaneously. In addition, agents of this type need to make decisions based on a large numbers of variables:

Positions and states of agents

Emergent patterns drawn from environment information

History of past actions

Results of past actions

Uncertainty of environment information

Goals received from higher-level agents

The decision-making task of these agents is significantly more complex, and if based on the traditional heuristic and procedural methods is extremely difficult to develop as it results in staggered logic-gates, often requires evaluating boolean logic decisions based on thresholds taken from non-discreet variables which becomes very hard to balance, and is not ideally suited to real-time operation as decisions tend to flicker when variables are close to threshold levels. We have therefore designed a more suitable decision-making model for complex agents.

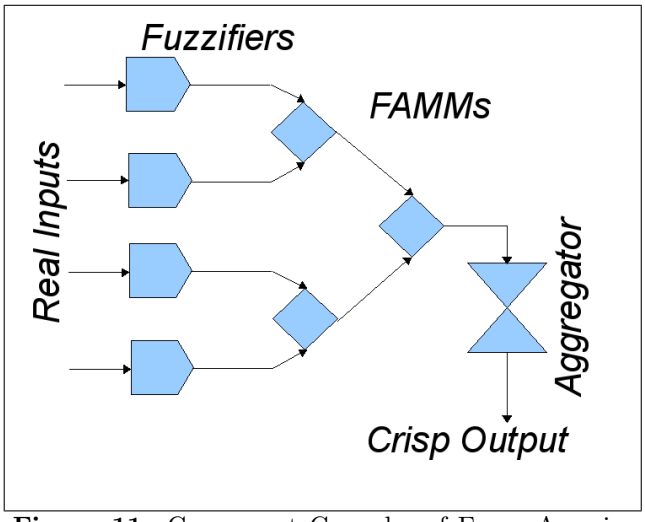

Figure 11: Convergent Cascades of Fuzzy Associative Memory Matrices for agent goal-task matching.

The Fuzzy Controller approach to decision making was originally developed to aid robot soccer strategy,

to take advantage of its very high degree of flexibility. It is possible to incorporate a multitude of input variables into a Fuzzy Strategy Layer with no added complexity to the decision making process. We no longer have to consider multi-dimensional decision arrays, and can sensibly interpolate unknowns by aggregation of Fuzzy outputs. If we analyse pairs of real inputs using Fuzzy Associative Memory Matrices, we can pass the Fuzzy Outputs (without defuzzification) as inputs to subsequent Fuzzy Associative Memory Matrices. Figure 11 illustrates this approach.

We can see that would be possible to analyse a very large number of input variables in this manner, and that due to the nature of the Fuzzy process, that this would be very fast and consume very little memory because we must only consider one 2d FAMM at any one time. The inputs would converge to a final 2d FAMM, and a single, crisp output value is produced through the aggregation procedure.

The critical task of the agent architecture is matching goals to actions, and we simply use a Fuzzy Controller module for this so that it can be left up to the designer what type of logic to use. The advantage of using fuzzified outputs of FAMMs as binary inputs to other FAMMs to make complex multi-variable decisions based on real-world data presents to a human agent designer a very clear, easy to understand and easy to adjust decision-making goal-action matching mechanism.

6.4 Coordinating Agents

Figure 12 illustrates a typical agent coordination problem. In this example, we are dealing with a battlefield simulation, where two tanks are controlled by cooperating agents. The first agent is at map index (C, 7), and the second agent at index (E, 9). The agents each have a limited model of their environment, covering 9 map cells, but they are in close enough proximity to each other that there is some overlap, which is indicated with a hatching pattern. In a robot system one of the agents would decide that it is the leader based on the order of a unique identifier, and then initiate the spawning of a coordinating agent on its own internal hardware, which might then be further distributed between agents to increase redundancy. In the illustrated example, however, we are dealing with a simulated battlefield, so we can simply designate a fixed chunk of system resources to a new coordinating agent. If each agent's system knows how to generate a new coordinator, then no new hardware would be required, and a coordinator could be quickly rebuilt if the hardware supporting it is destroyed or disabled.

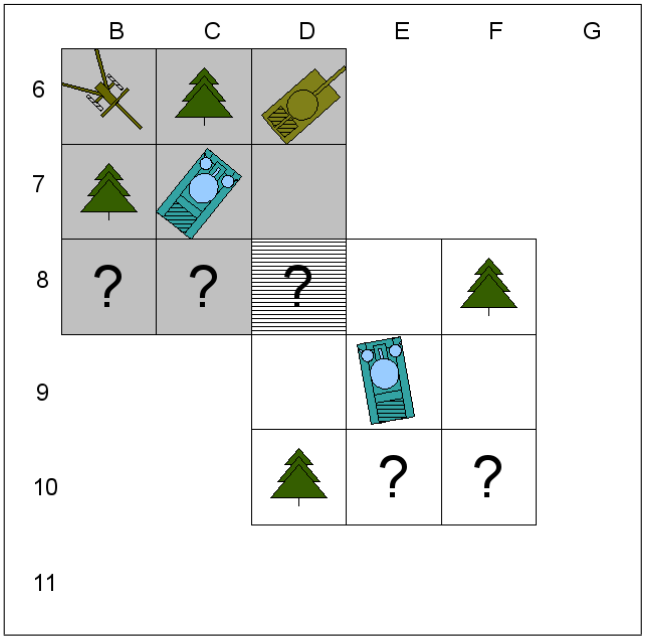

Figure 12: Agents cooperating over a shared world model.

Let us assume that the agent at position (E, 9) in Figure 12 has created a coordinating agent. The agent at index (C, 7) has spotted two trees, an artillery piece, and a hostile tank, so it communicates the map index and type for each of these features in a *field report* to the coordinating agent. The first agent does not know anything about the terrain behind it, which is indicated on its map by question mark characters. The agent at position (E, 9) has spotted two trees, and sends this information to the coordinating agent. It has also observed position (D, 8), which was unknown to our first agent, and so it communicates this information the coordinating agent also. We can assume that index (D, 8) has been confirmed to be featureless terrain, and so this then sent to the first agent, which can then reduce its 'wariness' or threat rating on this side of the vehicle, and concentrate more of its defences in the direction of the spotted threats.

By this stage the coordinating agent has built a map modelling the environment, with features covering much of the area between (B, 6) and (F, 10). The coordinating agent has time-stamped the information as it is received from the tank-controlling agents, which it considers to be its sensors. In a common peer-to-peer type approach, where cooperation is usually defined loosely as a sum of individual actions, both tank agents could have swapped environment information between each other and then made independent decisions; the first agent would probably have retreated

or attacked the artillery piece, which was more threatening to it than the tank, and the second agent may have moved to engage the hostile tank. The outcome of these individual decisions would have meant that the hostile tank had more time to move into a better defensive position, and move it's heavier armour to face the second tank and minimise its vulnerability, essentially evening the odds between forces. Under our architecture, however, the coordinating agent has identified that the key target is the hostile tank, and directs both of the cooperating agents to attack it from two sides - maximising the chances of destroying it, although putting the first tank at greater risk. The overall outcome, however would then move the advantage further to the side of the cooperating agents; a tactical decision that could never have been made by the 'selfish' model of peer-to-peer cooperation, which is clearly not able to make small sacrifices for the greater good of the group, or indeed any sort of higher-level tactical decision.

In other sorts of environments, cooperating agents can be used to exchange a large variety of environment information in the same manner; map cells can be used to store obstacles, terrain heights, terrain types or conditions, vehicle locations, agent locations, stages of planned routes of other agents (to help avoid conflicting agent paths), and a variety of role-specific information that lends itself to be discretised at the same resolution as the environment map. Retaining the simplicity of the environment maps is the key to their usefulness, as they can then be easily and quickly looked-up and communicated between cooperating agents.

6.5 Redundancy

We have developed our agent architecture with a lot of redundancy in mind. This is particularly useful for a cooperating society of agents with unreliable communication, robots prone to battery or equipment failure, for agents distributed over a number of physical machines, or for agents that can literally be destroyed.

Distributed agents should have some redundancy mechanism for regenerating lost modules on new hardware. The coordinating agents, of course can be easily constructed to be flexible in regards to the agents that it coordinates. Depending on the role for which they are intended, the coordinating agents could even be designed to dynamically absorb coordination of agents as they move into our out of communication range based on some sort of resource-discovery mechanism [32]. Loss of lower-level agents should not affect the coordinating agent. When all lower-level agents are lost, then the coordinating agent should collapse itself and inform any higher-level agents.

The bottom-level mobile agents should be able to operate autonomously, even if no coordinating agent is present. If a group of cooperating agents loses contact with their coordinating agent, or if the mobile agent hosting the coordinating agent is destroyed, then the mobile agent with the highest unique identifier should generate a new coordinating agent, with only short term loss to environment data, but a possible loss of history information. This system of redundancy remains true for higher level agents as well; if a group of level 1 coordinating agents can make contact then the coordinating agent with the highest unique identifier value (inherited from its host) can designate a bottom level agent to host a level 2 coordinator.

7 Summary and Directions

We have reviewed a range of existing architectures for control of autonomous agents and robots, specifically relating to *swarms* of agents that inter-communicate. We have discussed existing protocols of communication used by agents, and we have presented our own proposal for a novel agent middleware.

We are moving towards a complete architecture for autonomous agents that communicate, cooperate, and can be coordinated by a hierarchy of special coordinator agents. Because of the modular architecture based on independent but communicating components these new agents can be distributed over multiple cores and even geographically separated machines, which means they can be built with commodity hardware, or make use of existing idle resources.

This new architecture introduces a more effective model for multi-agent cooperation, and allows not only tactical decisions to be made in a coordinated manner, but enables cooperative actions that benefit the objectives of the whole group, rather than simply serving a large number agents' individual goals. This architecture is modular, easy to customise, applies to a large range of agent societies, and introduces distributed agents and redundancy to agent components.

Theoretically, 'factory' coordinator agents could be created like a virus (in the sense that it is not dependent on any physical machine itself but can exist in a transient sense within a communication framework) that have the role generating agents from pieces, using existing resources that it discovers on the fly - true virtual machines. Grid-enabled agents of this type would then provide massive redundancy to agents, so

that even if its actuators and sensors were disabled the agent could resume operation by communicating with new resources. For example; an automobile controlling agent that for some reason can no longer communicate with its cameras and range finders might be then be connected (by the factory agent) to the cameras of nearby vehicles and road-side sensors and continue operation with all of it's existing modules for steering and obstacle avoidance intact, successfully continuing navigation.

As agents become more resource-intensive the need to distribute agents grows, particularly with real-time agents that require large chunks of CPU time and process several tasks. Commodity hardware is also becoming more distributed - multi-core desktop machines, grid resources. A modular approach lends itself to a distributed system, and to some degree can be designed to work asynchronously. In this way the agent can exist within the Internet or grid, as long as it knows which modules (or resources) to communicate between. Grid-enabled and distributed intelligent agents may be the next evolution in this architecture, seamlessly taking advantage of any resources that are made available over complex networks whilst retaining the functionality of a cohesive agent machine.

Acknowledgments

It is a pleasure to thank: H.A. James, C.J. Scogings; M.J. Johnson; N.H. Reyes, A.L. Barczak; D.P. Playne; G.K. Kloss; S. Gordon and others who have worked with us on agent and robot control systems and with whom we have had many useful discussions.

References

[1] Thanngiah, S.R., Shmygelska, O., Mennell, W.: An agent architecture for vehicle routing problems. In: Proc 2001 ACM Symposium on Applied Computing, Las Vegas, USA (2001) 517–521 ISBN 1-58113-287-5.

[2] Settembre, G.P., Scerri, P., Farinelli, A., Sycara, K., Nardi, D.: A decentralized approach to cooperative situation assessment in multi-robot systems. In: Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008). Number ISBN:978-0-9817381-0-9, Estoril, Portugal (2008) 31–38

[3] Wikipedia: Kevin Warwick. Web Sites: `en.wikipedia.org/wiki/Kevin_Warwick` (2002)

[4] Sellem, P., Amram, E., Luzeaux, D.: Open multi-agent architecture extended to distributed autonomous robotic systems. In: SPIE Aerosense'00, Conference on Unmanned Ground Vehicle Technology II. (2000) Orlando, FL, USA.

[5] Brooks, R.: A robust layered control system for a mobile robot. IEEE Journal of Robotics and Automation **RA 2** (1986) 14–25

[6] Desouza, G.N., Kak, A.C.: A subsumptive, hierarchical, and distributed vision-based architecture for smart robotics. IEEE Transactions on Robotics and Automation **34** (2004) 1988–2002

[7] Andre, D., Russell, S.: State abstraction in programmable reinforcement learning. Technical report, Univ. California at Berkeley, USA. (2002) UCB/CSD-02-1177.

[8] Levesqu, H., Reiter, R., Lesperance, Y., Lin, F., Scherl, R.: Golog: A logic programming language for dynamic domains. Journal of Logic Programming **31** (1997) 59–84

[9] Hawick, K.A., James, H.A., Scogings, C.J.: A virtual prolog approach to implementing beliefs, desires and intentions in animat agents. In Zhang, S., Jarvis, R., eds.: AI 2005: Advances in Artificial Intelligence - Proc 18th Australian Joint Conference on Artificial Intelligence. Number LNAI 3809 in CSTN-022, Sydney, Australia, Springer (2005) ISSN 0302-9743, ISBN 3-540-30462-2.

[10] Soler, J., Julian, V., Carrascosa, C., Botti, V.: Applying the ARTIS Agent Architecture to Mobile Robot Control. In: Proc. Int Joint Conf 7th Ibero-American Conf on AI:Advances in AI. Volume LNCS 1952. (2000) 359–368 ISBN 3-540-41276-X.

[11] Firby, R.J.: The rap language manual. Technical report, University of Chicago, Chicago, Il, USA. (1995) Animate Agent Project Working Note AAP-6.

[12] Thrun, S.: A framework for programming embedded systems: Initial design and results. Technical Report CMU-CS-98-142, Carnegie Mellon University, Pittsburgh, PA, USA. (1998)

[13] Horswill, I.: Functional programming of behaviour-based systems. Autonomous Robots **9** (2000) 83–93

[14] Wilson, S.W.: The animat path to AI. In Meyer, J.A., Wilson, S., eds.: From Animals to Animats 1: Proceedings of The First International Conference on Simulation of Adaptive Behavior, Cambridge, MA: The MIT Press/Bradford Books (1991) 15–21

[15] Hawick, K.A., Scogings, C.J., James, H.A.: Defensive spiral emergence in a predator-prey model. Complexity International (2008) 1–10

[16] Scogings, C., Hawick, K.: Altruism amongst spatial predator-prey animats. In Bullock, S., Noble, J., Watson, R., Bedau, M., eds.: Proc. 11th Int. Conf. on the Simulation and Synthesis of Living Systems (ALife XI), Winchester, UK, MIT Press (2008) 537–544

[17] Hawick, K., Scogings, C.J.: Resource scarcity effects on spatial species distribution in animat agent models. In Grigoriadis, K., ed.: Proc. IASTED International Conference on Environmental Modelling and Simulation, 16-18 November, Orlando, USA. Number CSTN-059, Orlando, USA. (2008) 284–289 ISBN 978-0-88986-777-2.

[18] Hawick, K.A., James, H.A., Silis, A.J., Grove, D.A., Kerry, K.E., Mathew, J.A., Coddington, P.D., Patten, C.J., Hercus, J.F., Vaughan, F.A.: DISCWorld: An environment for service-based metacomputing. Future Generation Computing Systems (FGCS) **15** (1999) 623

[19] Mathew, J., James, H., Hawick, K.: Development Routes for Message Passing Parallelism in Java. In: Proc. of the AM Java Grande 2000 Conference. (2000) 54–61 DHPC-082.

[20] Leist, A., Hawick, K.: Small-world networks, distributed hash tables and the e-resource discovery problem in support of global e-science infrastructure. Technical Report CSTN-069, Massey University (2009)

[21] The Bluetooth Special Interest Group: The Bluetooth Core Specification. `www.bluetooth.com` (2002)

[22] Hawick, K., James, H.: Personal communications and mobile decision support. Technical report, Adelaide University (2002) DHPC-113.

[23] Dallas Semiconductor Corp.: Introducing Tini (2002)

[24] Packard, H.: iPAQ Handheld Mobile Devices (2008)

[25] Maravillas, E., Reyes, N.H., Dadios, E.P.: Hybrid fuzzy logic strategy for soccer robot game. Journal of Advanced Computational Intelligence and Intelligent Informatics **8** (2004) 65–71

[26] Gerdelan, A., Iskandar, D., Djohar, A.F., Reyes, N.: Utilising the Hybrid Fuzzy A* Algorithm in a Cooperative Multi-Agent System. In: in Conference Program and Abstracts of the 4th Conference on Neuro-Computing and Evolving Intelligence (NCEI06) and 6th International Conference on Hybrid Intelligent Systems (HIS06). (2006)

[27] Gerdelan, A., Reyes, N.: Synthesizing Adaptive Navigational Robot Behaviours using a Hybrid Fuzzy A* Approach. In: Advances in Soft Computing: Computational Intelligence: Theory and Applications. Springer, (Berlin & Heidelberg) (2006) pp. 699–710.

[28] Gerdelan, A., Reyes, N.: A Novel Hybrid Fuzzy A* Robot Navigation System for Target Pursuit and Obstacle Avoidance. In: Proceedings of the First Korean-New Zealand Joint Workshop on Advance of Computational Intelligence Methods and Applications. Volume vol.1., Auckland, New Zealand (2006) pp. 75–79

[29] Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall (1995)

[30] NASA: Peer-to-peer human-robot interaction project (2006)

[31] Carnegie Mellon University - Field Robotics Center: Lunar Rover Initiative (2008)

[32] Rana, O.F., Bunford-Jones, D., Hawick, K.A., Walker, D.W., Addis, M., Surridge, M.: Resource discovery for dynamic clusters in computational grids. In: Proceedings of the 15th International Parallel & Distributed Processing Symposium. (2001) 82