A Project to Develop a Distributed, Multi-Agent Communications Architecture Using Message Feedback

Richard Schilling
Cognition Group, Incorporated
Seattle, Washington 98108
schilling@cognitiongroup.biz

Abstract

This paper reports on an ongoing research project that is aimed at developing an intelligent agent messaging architecture that can support the development of large multi-agent systems for use in modeling, simulation, and practical business applications. The work has resulted in the definition of an agent messaging architecture defined by specific message delivery characteristics. Agents function as an intelligent agent ecosystem that spans multi-node computing clusters and communicate in a way that mimics naturally occurring biofeedback mechanisms.

Keywords

intelligent agent, biofeedback, CORBA, federation, communication, biosimulation

Introduction

There is a large body of literature documenting the application of intellligent agents to solve higher level problems such as knowledge representation (Picard & Gleizes 2002), beleivability (Bates 1997), and distributed content management (Zhang & Lesser 2004). Multi-agent systems applied to business problems such as supply chain management (Hillersberg *et al.* 2004) are also fairly common. Approaches to agent communications and agent organization based on newer network architectures such as wireless networks like Shah, Nixon and Ferguson propose (Shah, Nixon, & Ferguson 2004) are less commonly found. However, in most of the literature researched for this paper, the agent systems discussed were developed toward a narrowly focused problem. And very little attention was paid toward developing a general messaging framework that utilizes the capabilities of networking technologies developed after the year 2000.

No literature was found that made attempts to simultaneously deal with higher level knowledge representation, agent communication, agent organization, and practical modeling of real world processes. The focus of this research project is to attempt the development of a single, multi-agent architecture and communications framework that gives agent developers the means to deal with all of these issues. The resulting multi-agent architecture, agent feedback messaging, meets this goal by separating intelligent agents from their message communication infrastructure.

Scalability in agent communication is achieved by relegating the task of agent communications to a communication infrastructure. The communications infrastructure runs in its own thread space at the application layer of the network protocol. Agents are not allowed to control other agents or invoke their methods directly. Each agent resides in its own thread space and communicates with other agents by interacting with the communications infrastructure.

How the communication infrastructure sends messages between hosts is an implementation level detail. The example implementation presented in this paper uses the Internet Inter-Orb Protocol (IIOP) to transmit messages between hosts. The federation design pattern for agents catalogued by Hayden et al (Hayden, Carrick, & Yang 1999) is the closest example of a similar approach, however in the present project federation is used only to transmit messages between multiple hosts running agents, and not between agents directly.

The agent communications architecture presented has become an effective way to develop arbitrary ecosystems of cooperating agents and enforces message delivery characteristics that: a) eliminate dependencies between agents; b) removes the need to include network code in agents such as sockets or datagram programming; and, c) allows the physical layer of the network to change without affecting agent functionality.

Agent Communications and Feedback

The approach to communications taken on the agent feedback messaging project is a marked departure from the traditional agent-level point-to-point communications. The departure was necessary to avoid problems associated with large scale agent design that are well documented. Becker et al (Becker, Lesser, & Zilberstein 2005) make it clear that when agents are responsible for delivery of messages directly, the resulting myopic communication designs ultimately sacrifice overall system performance. Even when highly scalable federated agent communication approaches using remote method invocation (RMI) are used as in the Cougaar project (Cerys, Rozga, & Berliner 2006), the programmer must still manage the complexities of determining which agents communicate with other agents and when.

Hayden et al (Hayden, Carrick, & Yang 1999) properly associate these issues with two desirable features of agent communication systems: a) allowing agents to serve as both client and service-provider, and consequently b) allowing agents to use a complete agent network, a communication network between agents that forms a complete graph.

Agent feedback messaging attempts to provide these two features and deal with the associated issues by completely removing the burden of message addressing and delivery from the sending agent. The only way an agent is allowed to communicate with other agents is by inserting messages blindly into a constantly moving circular message flow, or stream. No destination agent is specified by the sender at any time. It is the responsibility of the receiving agent to detect messages it needs in the message stream and process them. This limits an agent's communication tasks considerably and removes the need to hard-code the destination of a message.

Figure 1 provides a graphical representation of agent feedback messaging. A circular message stream is the central feature of agent feedback messaging. When an agent inserts a message into the message stream it is attempting to incite other agents to perform some activity and, in turn, input their own messages (feedback) into the message stream. The feedback messages are then picked up by the agent that sent the first message. The number of feedback messages created in response to a given message is up to the agent developer. Agent feedback messaging does not require feedback messages to be generated in response to any message.

Every agent is assigned its own unique point of interaction with the message stream from which it can read messages from other agents and insert new ones. The circular message stream, operated by its own set of threads, simultaneously retrieves new messages from all agents. Each message is delivered by the message stream to all other agents in succession and is eventually returned to the originating agent. As each agent is presented with a message it determines if the message applies to it before the message stream sends the message to the next agent.

The basic interaction that occurs between all agents is depicted in Figure 2.

Reducing Agent Coding

In terms of writing agent code, the overall effect of the approach described is a dramatic reduction in the coding required of the agent developer. In a system where any agent is allowed to communicate with any other agent, the possible communication paths form a complete directed graph, and can be expressed as

$$p = n(n-1)$$

where n is the number of agents in the system and p are the possible number of communication paths. If an agent is required to obtain a handle or reference to another agent in order to send a messge, then the possible number of lines of code that must be written for all agents to send one message can be expressed as

$$c = ln(n-1)$$

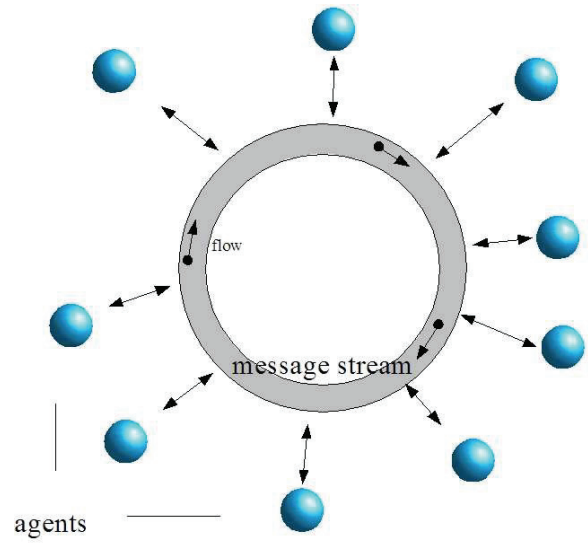


Figure 1: Agent Feedback Messaging Architecture

where l is the number of lines of code required for an agent to send one message and c are the total lines of code for all agents. c does not include any lines of code written into agents to receive messages.

When agent developers are not required to hard code message destinations into the agents themselves, the overall effect is that the number of lines written for all agents to send one message can be limited to

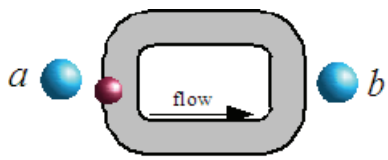
$$c = ln$$

The project has revealed that this lower burden on agent coding can be achieved without sacraficing reliability or predictability in message delivery if certain rules are applied to the communications layer. The rules that have been settled upon, as of this writing, allow the agent developer to make some concrete assumptions about how messages are delivered:

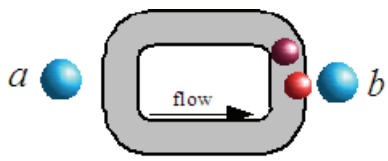
1. an agent is never aware of which agents will receive its messages before sending them.
2. an agent is not required to respond to every message message presented to it.
3. only the agent that creates a message has the right to destroy it.
4. any message an agent creates will be delivered to all other agents in the system before being returned to it's creator.
5. a message is presented to all agents in succession, rather than being broadcast.
6. for a given set of active agents the order of agents the message is presented to does not change.
7. for a given agent in a set of active agents, the first agent to receive its messages is unique.

8. a message can be assigned read/write policies, so that agents can add or modify message information as it passes by.
9. the order in which messages are inserted into a stream relative to other messages is maintained.
10. the order in which a given agent sees messages reflects the relative message order in which it was inserted into the message stream.
11. the relative rate of travel between messages in the message stream is zero - messages always travel in the message stream at the same speed.
12. the amount of time it takes for a message to circulate to all agents is independent of the amount of time it takes for any agent to process the message.

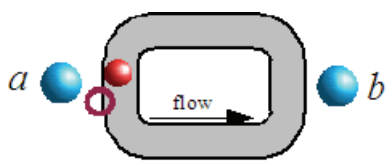
Any agent messaging system that imposes these rules on message delivery can also be said to be an agent feedback messaging system.



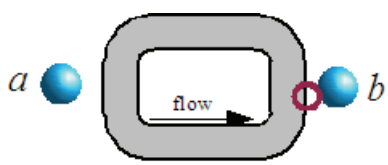
(a) Agent *a* dispatches a message.



(b) Agent *b* receives the message and dispatches a response (feedback).



(c) Agent *a* receives its original message and destroys it. It then receives the response message from agent *b*



(d) Agent *b* destroys the response message it created.

Figure 2: Basic Agent Interaction

The time for a given response to be returned to an agent is undefined, and is viewed as mostly a hardware performance issue. However, the rules imposed on the message delivery infrastructure guarantee that messages and their associated response messages (feedback) are always delivered in the proper order: a message's delivery always preceeds the delivery of feedback messages. It is therefore possible to guarantee that message/feedback delivery occurs in a prescribed order under a simulated timeline.

Agent feedback messaging also allows for the special case of a message to be modified directly as it is processed by other agents. This is possible because the only time a message is destroyed is when the sending agent determines that it should be, and only after a message has been delivered to all other agents at least once. Therefore all messages have a read/write policy. Some messages contain data that can be only read (read only policy), while others allow agents to add or change information (read/write policy), and yet other messages require agents to add information (write required policy). Write required policies are used to solicit information from agents that can be supplied immediately, such as the agent's name or the results of a simple operation. The read/write policy of a message is determined by the developer of the agent that sends the message.

While other agent communication infrastructures emphasize the delivery of messages between agents in a broadcast fashion (multicast delivery) (Shah, Nixon, & Ferguson 2004), agent feedback messaging delivers each message to agents one agent at a time (linear delivery). Both approaches have advantages, but the author has found that when the combined processing time attributable to message delivery itself, message filtering (deciding if an agent should get a message), message sharing (semaphore locking/unlocking), and response processing (feedback) are taken into account, linear delivery provides advantages that outweigh any percieved loss of performance over multicast delivery. Furthermore, because agent feedback messaging requires each agent to be executed in its own thread space and share data through inter-thread shared memory structures, the complexity associated with managing method call stacks for a large number of agents is reduced.

The designer of an agent determines what messages the agent must respond to. To handle the case where an agent is required to indicate it has no response to a message, the agent can a) create and send another new "no response" message back, or b) add the "no response" data to the message it received, or c) do nothing. The third option, do nothing, is a valid "no response" action because this allows for intentionally delayed responses to occur.

Agent Design Methodology

With a generic "fire and forget" message delivery system in place, it was discovered that agent design methodology could handle scenarios where an agent serves as both a client and service provider, even in the case where an agent fills both roles for its own requests.

In order to develop agents, a basic set of four questions is answered for each agent:

1. What messages are the agent required to listen for?
2. What is the read/write policy of the message?
3. What actions are performed in response to the message?

4. What messages does the agent send, and when does it send them? Note that the target agent of a message is never defined by the sender.

Message	StartSimulation	HeartBeat	NeedMoreProtein	NeedMoreNourishment	Eat	ProteinMessage	NourishmentQuantity
Agent							
Human	S	R			R	S	
Stomach	R	R	S	R		R	S
Heart	R	S		S			R
Left Arm	R	R		S			R
Right Arm	R	R		S			R
Left Leg	R	R		S			R
Right Leg	R	R		S			R
Brain	R	R	R		S		R
Message Purpose	Tells all agents to start simulation.	Simulates heartbeat to mark the simulation interval.	Indicates that more protein is needed.	Indicates that more nourishment is needed.	Instructs the person to eat.	Indicates that protein (food) has been eaten.	Used to deliver nutrients throughout the body.

S = message sent by organ
R = message received by organ
blank = message ignored by organ

Table 1: Biosim Message Delivery Matrix

With these questions answered for each agent, a message delivery matrix can then be developed for an entire system of agents. See Table 1 for an example.

Keeping a master catalogue of agents and agent messages is also useful in the development of systems that use agent feedback messaging. This makes it possible to develop systems that use existing messages and agents designed by other developers. When existing agents in the catalogue do not meet a designer's needs, existing messages from the catalogue can still be incorporated into new agent designs.

Using Agent Feedback Messaging for Biosimulation

The first test case of AFM was to develop a very simple simulation involving biofeedback. The source code for the simulation is packaged as part of the AFM distribution. The idea to use intelligent agents to simulate natural biofeedback mechanisms was the author's original inspiration for agent feedback messaging when work to develop the approach started in the early 1990's.

In this simulation, agents are created to simulate the distribution of nutrients throughout the body and the interactions that take place between various body parts to signal to a person that he needs to eat more food. Agents represent the brain, heart, stomach, legs, and arms. The decision to eat and the act of eating are both managed by a single "Human" agent.

The simulation demonstrates many aspects of agent feedback messaging, however the top five are:

1. no agent knows what agents are responsible for handling messages it sends.
2. all responses to messages can be in the form of other messages generated by different agents (feedback messages).
3. messages can contain values that get updated as a message is passed from agent to agent (e.g. a message representing a resouce can be "depleated" as it passes between agents).
4. the order in which agents are instantiated in the system can be used to ensure that some agents receive messages prior to others.
5. it is possible to simulate biological processes when all responses to messages are received in a delayed manner (non real-time).

Table 1, a message delivery matrix, identifies the agents in the simulation and the messages each agent generates and receives.

The simulation starts when the Human agent creates all the other agents and then sends a StartSimulation message. No reply is expected from this message. The Heart agent simulates the passage of time by generating a HeartBeat message at regular intervals. In a real simulation the Heart agent could be programmed to generate a HeartBeat message once it determines that all simulation activity associated with a single heart beat has completed. The Heart agent would determine this by listening to messages produced by other agents.

As time passes, the arms and legs use up nutrients they have stored. Each appendage keeps track of how many nutrients it has stored in its internal state and each uses nutrients at a different rate. As each appendage runs out of nutrients it dispatches a NeedMoreNourishment message requesting more. The feedback appendages expect to receieve at some point in the future is a NourishmentQuantityMessage. Each appendage continuously dispatches NeedMoreNourishment messages until it receives enough NourishmentQuantityMessages to satisfy its internal requirements to have a surplus of nourishment in storage.

The Stomach agent, upon seeing NeedMoreNourishment messages, dispatches a NourishmentQuantityMessage; one

for each NeedMoreNourishment message it receives. Stomach keeps track as part of its internal state how much nourishment it can release. As the supply of nourishment in the stomach depletes, the Stomach agent begins to request more protein (which it uses to create nourishment) by dispatching NeedMoreProtein messages. Like the appendages, the Stomach simply expects to see a ProteinMessage at some point in the future, but it does not know when that message will arrive. NeedMoreProtein messages are dispatched in regular intervals until it has the protein it requires.

The Brain agent, which is responsible for listening for NeedMoreProteinMessages, signals that more food must be eaten by dispatching an Eat message. As with all the other body parts, Brain does not know what agent actually is responsible for responding to this message. The more NeedMoreProteinMessages Brain receives, the more Eat messages it dispatches.

The Human agent, an agent that simulates the complicated process of actully eating, responds to Eat messages by dispatching Protein messages. The Stomach agent in turn picks this message up and interprets it as a signal that it should generate more nourishment. Each Protein message the Stomach agent receives causes Stomach to add a specific quantity of nourishment available to its nourishment store, which it can dispatch at a future time whenever it sees a NeedMoreNourishment message.

Agent feedback messaging allows messages to be modified as they are passed from one agent to another. This makes it possible for one message to represent a resource that changes as it circulates among agents. In this simulation, as NourishmentQuantityMessages are read by each agent, the agent "extracts" nutrients from the message by reducing the amount of nutrients the message has. When the NutrientQuantityMessage runs out of nutrients it is no longer valuable and will be ignored by other agents.

The simulation also illustrates that the order of agents on the message stream can be used to enforce priorities in message consumption. Downstream agents closer to the Stomach agent receive NourishmentQuanityMessages first, and therefore have the first opportunity to use the resources in the message prior to downstream messages.

Dependency Issues

Before discussing an implementation of agent feedback messaging, it will be helpful to describe how agent feedback messaging can help developers address dependencies in their programs. Although biological simulation was the original intended use for agent feedback messaging, the author has also found that it can be used to work around dependencies introduced into programs and make them more managable. As Horling et al (Horling *et al.* 2006) illustrates, the task of managing the code behind agent behavior and communication can increase with the number of distributed autonomous agents involved. Therefore, eliminating dependencies between agents such as hard-coded destination agent names and programming logic that spans agent boundaries is a priority of the agent feedback messaging project.

Many of the issues surrounding managability appear to be related to the need to ultimately execute the code behind agents and communicate data between them in a more or less linear fashion. This could be said to be a side-effect of the Turing model. But applications reducable to a Turing Model do not always map well to higher level processes (biological or human defined) that occur in a non-linear fashion. The Turing Model necessitates that one piece (or block) of code be evaluated before others. In general, when one block of code is required to be executed before another block of code we can say that the execution of the second block of code is dependent upon the first block. We can call this a dependency.

There are many reasons why dependencies exist, but they can greatly impact how computer code itself is written and maintained. Dependencies can be imposed on a program by the the language it is written in as well as the architecture it is executed on, even if the dependency is not required by the programmer. An example would be a single processor machine that can only execute one instruction at a time. Dependencies can also be determined by the higher-level design of the application such as the system's functional policies (e.g. a security requirement), a business process the application must support, or a biological processes the application might simulate. For the purposes of discussion it is helpful to categorize dependencies into three general categories.

Code Dependencies

Code dependencies describe the order of execution of code which can be attributed to the design of a computer language. Programmers always rely on code dependencies to make sure a program executes its instructions in a prescribed order. Code dependencies also exist whenever the order of execution of machine level instructions is enforced simply because the language does not include syntatic features that allow parallel execution of routines. Parallel architectures and pipelining in processors helps alleviate this problem at the machine level, but introducing code dependencies even when they are not necessary is unavoidable in many cases.

Temporal Dependencies

A temporal dependency is the deliberate ordering of code block execution even though it is not imposed by a code or process dependency (described below). Temporal dependencies exist whenever one part of a program must be executed prior to another part in order to satisfy some goal. This type of dependency would most likely occur, for example, when one function can be executed either before or after another, but for some reason the programmer enforces a particular order. In a real-time system, temporal dependencies are strictly enforced since the functioning of the system relies on the execution of a routine at a pre-defined moment in time. In a non real-time system, temporal dependencies can be created by the timing of code excution to support well defined steps in a single business process (e.g. payroll processing). Temporal dependencies can also be created by simple requirements external to the application. For example, one part of a program may only be allowed to run at night for maintenance purposes, while another part of a program can be run during the day for routine operations.

Process Dependencies

A process dependency (or "process level" dependency) is the deliberate ordering of code execution at the application level to execute the steps in a process. Process dependencies can also be created to execute one code block (or perhaps an entire appliction) prior to another in order to meet some system policy. Process dependencies, for example, can be imposed to reflect the requirement of one business process (or simulated natural process) to terminate before another starts. For example, a business may require the execution of a data collection program prior to the execution of a data analysis program that operates on the collected data. Or, in a program simulating an ecosystem, a process dependency can be created to ensure that a rain simulation starts prior to the execution of a river flood simulation.

Whereas process dependencies describe the timing of code execution across application boundaries (in the *macro*-application sense), code and temporal dependencies describe the timing of code execution within a single application (in the *micro*-application sense).

Removing Dependencies

The three types of dependencies described above can make agent design and maintenance difficult when a programmer must deal with all three simultaneously. Agent feedback messaging is intended to help alleviate this issue by allowing a developer to focus on the process and temporal dependencies without having to worry about the code dependencies that are imposed by the underlying system. Temporal and Process dependencies are completely controlled by the agent developer through the following means: a) by specifying which agents produce messages and when; b) by specifying what feedback messages, if any, to generate as responses to messages; and, c) determining the order in which agents are assigned on the message stream.

Furthermore, each agent has its own call stack and runs in its own thread space independent of the other agents and the message communications infrastructure. Consequently, the developer does not have to worry about returning message results from an agent to the top of a function call stack before delivering it to another agent. Messages are passed, rather, by the communication stream in a producer/consumer fashion through the use of shared memory structures such as shared FIFO queues.

This better control over dependencies and agent feedback messaging's circular message stream helps the development of simulations by providing a way for an agent to dispatch both data and service requests to all other agents without having to keep track of which agents actually receive the message. The agent that sends a message doesn't even need to be the agent that responds to any replies to the messaage. This allows an agent developer to have complete control over the granularity of agents involed in a simulation.

In their simplest forms, system designs based on agent feedback messaging allow each agent to deliver messages in a "fire and forget" fashion. Messages are created by an agent and completely released into the communications stream. They are only seen again by the agent that created them after all the other agents have seen them. The agent also does not wait for an immediate response, but rather does other processing while it waits for its messages to be returned.

AFM: An Implementation of Agent Feedback Messaging

AFM is an implementation of agent feedback messaging and serves to demonstrate how agent feedback messaging can be implemented in a practical solution. AFM has also produced some useful benchmarks which are described below that could be used as a basis of comparison to other agent communication infrastructures. AFM is available for download at no charge at www.cognitiongroup.biz/AFM.

AFM puts each agent into its own thread space. Each agent utilizes a number of lightweight processes (four threads per agent) that utilize semaphores and shared memory space (FIFO queues) to pass messages from one agent to the next. The agent thread space and mechanisms for moving messages between agents is depicted in Figure 3.

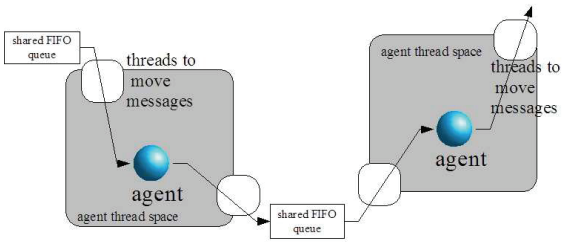


Figure 3: Agent Thread Spaces

AFM's design requires each host to pass agent messages exactly one time to all agents on a given host before sending them on to another host. The host that receives each message, in turn sends the message to all agents running on that host, and then passes the message to the next host, and so on. After each message has been presented to all agents running on all other hosts, it is returned to the originating agent.

The passing of messages between hosts is depicted in Figure 4. This design in effect creates a large, circular, multi-host, message stream that all agents have access to. In order to deliver messages between hosts, AFM uses CORBA valuetypes to send marshalled message data via IIOP. CORBA valuetypes are used for two reasons. First, they allow message objects to be passed by value from host to host without involving agents directly in network communications. Second, message valuetypes support the passing of entire graphs of messages while preserving shared values in the graph. Pages 54 and 55 of the book *Java Programming With CORBA* (Brose, Vogel, & Duddy 2001) describe this mechanism. This approach adds to the AFM implementation the ability to define message heirarchies using both class inheritance and member encapsulation.

The FIPA Agent Messaging Transport Protocol (FIPA b) could be used to transmit messages between hosts as well as long as the use of the protocol is restricted to AFM's communications infrastructure and not programmed directly into agents. Transporting of valuetypes via the existing FIPA

IDL interface would most likely require some additional work however, such as extensions or additions to the FIPA IDL definition. In addition, parts of the protocol would also be ignored by AFM. That protocol requires in every message the designation of a destination agent to receive the message. In AFM it would not be required to specify the destination agent because it is the responsibility of an agent wishing to receive a message to catch the message. Consequently the field containing the destination agent would be ignored by AFM.

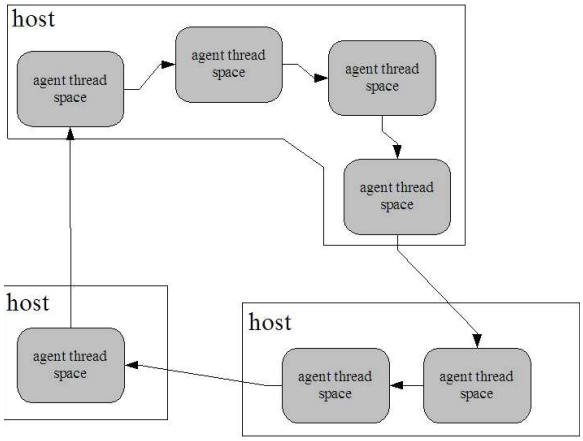


Figure 4: Message Passing Between Hosts

AFM is implemented in Java. There are several features in the Java platform that makes it an attractive platform for implmenting an agent feedback messaging system:

- robust shared memory object support and semaphores
- large thread count management with critical sections
- automatic garbage collection
- strong support for CORBA interoperability
- platform independence
- acceptance in the scientific community

Federated Agent Communications

Federated design architectures make it possible to implement agent feedback messaging as a scalable mesaging infrastructure where hosts running agents can be added and removed to the network dynamically. However, federated approaches have rarely been applied to the realm of intelligent agent communications. The FIPA Abstract Architecture (FIPA a) is the most notable exception.

The agent feedback messaging project attempts to open the door to effective federated designs in agent communications. Forslund, et al (Forslund, Smith, & Culpepper 2000) demonstrate that integrating the functionality of several stand-alone systems through federation is an effective way to integrate network based services across clusters of computing nodes and security domains. Hayden et al (Hayden, Carrick, & Yang 1999) also document some federated agent design patterns useful for building multi-agent systems. These are the approaches taken to implement the message communications infrastructure with the present implementation of agent feedback messaging.

Federation on the agent project was achieved by defining an interface for message communications between hosts using CORBA. Each host registers itself wth an object request broker so that other hosts can discover the messaging service at runtime. Because standard IIOP is used other non-AFM systems, such as datagram broadcasting systems and peer-to-peer systems can be used in conjunction with AFM.

IIOP is also not the only networking protocol that AFM can be implemented on top of. Since the networking protocol in use is completely transparent to the agents running in the system it can be replaced by something else entirely.

AGENT MESSAGING BENCHMARKS

The biological simulation presented has produced some usable application wide benchmarks that might be applied to all agent messaging systems.

idle message delivery time: the time it takes for one message to be delivered to all agents in the system and a result to be returned when the system is idle (no other messages being delivered). Can be expressd in revolutions per unit of time where the dispatch and return of one message is one revolution.

per agent thread count:the number of threads required per agent.

per agent memory usage: the average amount of memory used per agent.

messages delivery rate: the number of messages sent and successfully delivered per unit of time.

message throughput: the maximum numer of messages the communications infrastructure can deliver per unit of time.

Plotting these benchmarks against the number of agents in the system provide a useful operating envelope for an agent message delivery system.

Conclusions

The performance of agent feedback messaging, as implemented in AFM, is good enough to begin the creation of large scale simulations of both natural and man-made processes. The number of agents that can be reasonably instantiated on a single computer without a significant loss in overall performance is high enough to attempt agent based simulations of larger models, such as Forrester's production distribution system (Forrester 1962). In addition to continued work on the biosimulation example presented above, work has also begun on simulations of Forrester's production models as well as simulations of a biodiesel manufacturing operation.

Agent feedback messaging also has given the author the ability to reduce the amout of time managing code dependencies and spend more time managing the procedural and temporal dependencies that define a given simulation. The granularity of components smulated in a given model can also be manipulated relatively easily, with the number of

agents in a simulaton being inversly proportionate the to the size of object in the model being simulated. A large number of agents can be used to simulate the smallest elements in a model (e.g. cells in an animal), or a single agent can be used to simulate one large object (an abstract animal). During simulation development, coarse grained agents can be used to validate a simulation against broad parameters. Fine grained sets of agents can then be developed that deal with detailed parameter sets in a model. This provides enough flexibility to pursue bottom-up and top-down approaches to agent design simultaneously.

Efforts to integrate agent feedback messaging with external data sources have also resulted in opportunities to integrate intelligent agents with commonly used web services technologies.

References

Bates, J. 1997. The role of emotion in believable agents. *Communications of the ACM* 37(7):122–125.

Becker, R.; Lesser, V.; and Zilberstein, S. 2005. Analyzing Myopic Approaches for Multi-Agent Communication. In *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 05)*, 550–557. Compiegne, France: IEEE Computer Society.

Brose, G.; Vogel, A.; and Duddy, K. 2001. *Java Programming with CORBA, Third Edition*. New York, NY, USA: John Wiley & Sons, Inc.

Cerys, D.; Rozga, A.; and Berliner, J. 2006. A cougaar-based logistics modeling tool for highly adaptable military organizations. (online) http://cougaar.org/docman/?group_id=17, downloaded March 2006.

FIPA. Fipa abstract architecture specification. (online) http://www.fipa.org/specifications, downloaded May 2006.

FIPA. Fipa agent message transport protocol for iiop specification. (online) http://www.fipa.org/specifications, downloaded May 2006.

Forrester, J. W. 1962. *Industrial Dynamics*. Cambridge, MA.: Massachusetts Institute of Technology (M.I.T.) Press.

Forslund, D. W.; Smith, R. K.; and Culpepper, T. C. 2000. Federation of the person identification service between enterprises.

Hayden, S.; Carrick, C.; and Yang, Q. 1999. Architectural design patterns for multi-agent coordination.

Hillersberg, J.; Mooonen, H.; Verduijn, T.; and Becker, J. 2004. Agent technology in supply chains and networks. In Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT '04). Beijing, China, September 2004.

Horling, B.; Lesser, V.; Vincent, R.; and Wagner, T. 2006. The Soft Real-Time Agent Control Architecture. *Autonomous Agents and Multi-Agent Systems* 12(1):35–92. An earlier version is available as UMass Computer Science Technical Report 2002-14.

Picard, G., and Gleizes, M. P. 2002. An agent architecture to design self-organizing collectives: Principles and application. In *Adaptive Agents and Multi-Agents Systems*, 141–158.

Shah, S. W.; Nixon, P.; and Ferguson, R. I. 2004. On the use of ip multicast to facilitate group communication between mobile agents. *iat* 00:487–490.

Zhang, H., and Lesser, V. 2004. A Dynamically Formed Hierarchical Agent Organization for a Distributed Content Sharing System . In *Proceedings of the International Conference on Intelligent Agent Technology (IAT 2004)*, 169–175. Beijing: IEEE Computer Society.