Loom: A Development Framework for Distributed Intelligent Agents

Josh Brown

I. Introduction

Intelligent agents and artificial intelligence (AI) techniques are being developed and researched in nearly every discipline. Since the conception of AI a variety of applications have used AI techniques but only recently with the advent of data mining and other techniques has there been a surge in the community of developers and researches interested in advancing AI. There is an endless horizon of possibilities for AI. Currently AI is visible in everything from our microwaves at home to offshore oil platforms.

Already computer intelligence is making decisions for us. When we log into amazon.com[1] we are greeted with books and movies that we will most likely be interested in. We can buy vacuums that will intelligently clean our floors without our presence[2], and our vehicles have AI components that equalize braking pressure and turn the volume of our radios up to compensate for road noise at high speeds. Although this technology is just beginning to emerge in our every-day lives in limited applications and environments we will soon find that many of our decisions will be made by intelligent agents.

In the last five years there have been significant advances in computing capabilities that are influencing how AI is and will be developed. Although some standards are beginning to emerge, we find more applications of AI popping up every day and it is becoming apparent that these systems will benefit from, and in fact need to, communicate with each other in order to perform complex tasks. The necessity of having communication between intelligent agents in order to perform tasks is the key indicator that a distributed solution, or possibly middleware solution, is needed.

There have been some attempts at creating middleware for artificial intelligence in a very limited setting. The video game industry has several notable examples where attempts are currently underway to develop an AI middleware that for computer game developers. However these mostly focus on creating common connections between an intelligent agent and the interaction within a game, not making connections with other agents. There have been other attempts at creating frameworks for developing artificial intelligence with common interfaces and predefined constraints such as OpenAI[3] which attempts to generalize several AI paradigms. There is also the Open Agent Architecture[4] which is one current framework for distributed agents. Today's research is yielding many possible solutions and we cover some of these in the related work section.

The lack of a simple interface or development framework that lends itself to developing distributed agents will severely limit the capabilities of AI developers in the future, causing them to reinvent the wheel with every agent they develop. We believe that in order for intelligent agents to continue to develop and become more useful that the developers of agents will need to be able to write agents for a distributed environment. This distributed environment will most likely require a high degree of transparency to the developer of new agents, and almost complete transparency to the end user.

The AI community needs an open standard development platform in order to solve current problems and provide a basis for further progress in the field. A distributed system for intelligent agent development can alleviate current problems and avoid future hardship by providing this open and simple framework.

A. *Statement of purpose*

This research addresses the lack of a standard development environment for distributed AI that is easy to learn but is still extensible, maintainable, and freely available to the AI community. This research introduces Loom, a framework that lends itself to the development of distributed intelligent agents.

The purpose of this research is to design and develop a distributed agent system called Loom. The research starts by analyzing the specific requirements of intelligent agents and their developers. Next using these requirements as a guide we look at current techniques and learn what tools are currently available. Using these prior experiences and our defined requirements we outline an initial design for a new development environment, Loom. Once this initial design has been created we will test the distributed nature of the system. We mean that we will test both the ability for the system to work and communicate as a distributed system and the performance gained by the agents. We will also test how effective the development framework is by testing how easy it is to develop an agent ad how quickly a new developer can acclimate himself to the API and the system's methodologies.

B. *Scope of research*

The proposed research will be completed in several stages.

1) Define specific needs and requirements of all intelligent agents.
   Only after evaluating the requirements of intelligent agent applications and distributed system separately and then distributed agent applications can we successfully develop a standard development framework for the development of distributed agents. Once the requirements have been defined then we can proceed to evaluating current tools and solutions.

2) Analyze any existing tools and techniques for creating and developing distributed and non-distributed agents
   A careful evaluation and analysis of current tools was done to develop an understanding of the strengths and weaknesses of these tools. We look at several tools currently being used in distributed systems, artificial intelligence applications, and distributed agent applications. From this evaluation we were able to determine many useful features and functionalities important to the creation of Loom.

3) Initial design implementation of Loom based on requirements
   After completing an evaluation and determining key functionality as well as determining poor designs from previous work we create a list of features and

components for a new distributed agent model. Using this list as a base we create an initial model and then evaluate it against our design goals. This iterative design process evolves until we have a robust model that deals with our design goals. This model is the basis for the design of Loom.

4) Test with the development of real-world applications
We develop two test applications. One for the Virtual Reality Applications Center (VRAC) and one simple test case. These two test cases test the design of Loom and provide feedback that will hopefully lead to further iterative development of the development package.

These four stages are presented in this thesis in the following chapters:

- Chapter 2 covers background material and definitions of AI and distributed computing
- Chapter 3 satisfies stage 2 by presenting previous and related work in this area.
- Chapter 4 covers stage 1 by defining the requirement for the distributed agent system.
- Chapter 5 satisfies stage 3 by laying out the design and implementation of Loom
- Chapter 6 satisfies stage 4 of our research by observing real world applications and their implementations using Loom.

II. Background

A. The future of AI

AI has an interesting history and perhaps an even more interesting present, but perhaps the most exciting and most interesting time for AI lies in our future. Although we may never succeed at the Turing test, (see below) and we may not reach the level of computer intelligence in movies such as the Matrix we will see ever increasing intelligent agents helping us make decisions and making decisions for us.

Imagine a world where you never need to drive your car, an intelligent agent knows how to drive you to your destination in the shortest time using the minimal amount of fuel. Imagine for a moment having all of your groceries delivered to your door - ordered by your refrigerator when it noticed you were low on supplies. Imagine not having to wait in security lines at airports because intelligent agents can identify everything on people and in bags using sonar. These examples are all current research topics around the world with the goal of easing our day to day lives.

AI shows amazing potential, but there are many obstacles and challenges, both technically, socially, and even philosophically that impede progress. A debate over the social and philosophical issues involved in the creation of some intelligent agents would be interesting, however we will leave that debate for another paper and focus on the technical challenges.

B. What is AI

There is no well agreed to definition of what AI is. Russell and Norvig[5] create a taxonomy for categorizing AI into four distinct groups. The four categories are listed below with definitions of AI supplied from various sources for that categorization:

1) Systems that think like humans – “The exciting new effort to make computers think ... machines with minds, in the full and literal sense” [6]
2) Systems that act like humans – “The art of creating machines that perform

functions that require intelligence when performed by people" [7]

3) Systems that think rationally – "The study of mental faculties through the use of computational models" [8]
4) Systems that act rationally – "A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" [9]

These four categories vary along two main dimensions. Do the systems think or act, and are they pursuing human behavior or rational behavior? Let's briefly explore each of these categories.

1) Acting human : the Turing Test: Perhaps the most academically famous artificial intelligence test is the Turing Test, developed by Alan Turing in 1950. His test was designed to test if a machines qualified as intelligent. Turing defined intelligence by a computers ability to sufficiently fool an interrogator. Basically the *Turing Test* was to put a computer at one end of a teletype and a human at the other. After 5 minutes if the human could not tell if he or she were communicating with a human or computer then the computer passed the test.

2) Thinking humanly: The field of cognitive science is what this category has evolved into. Cognitive Science attempts to bring together computer models of AI and experimental techniques from psychology to try to construct theories of how the human mind operates.

3) Thinking rationally: The idea or concept of logic is the primary concern of agents that aim for thinking rationally. Rational thinking can be thought of as thinking correctly at all times or thinking perfectly, therefore logical connections between known entities can lead to new connections which can be seen in learning. For example, "Socrates is a man; all men are mortal; therefore Socrates is mortal."

4) Acting rationally: Acting rationally means attaining one's goals, given one's current status and desired outcome. We can define a *rational agent* as an entity that perceives a current state and acts based on that state to achieve a goal. This approach has many positive qualities. All the skills necessary to complete the Turing Test allow for rational actions. It is also more general than the logic based thinking rationally.

Because of the general nature of the acting rationally category it has come to the forefront in most current research as the end goal for an agent. Therefore, we determine just as Russell and Norvig do that no matter what the specific goal for an agent is, the more general aim of the agent in one form or another is to act rationally. In conclusion AI is the generation of rational agents.

C. Characteristics of Agents

As we determined in the last section AI is the generation of rational agents (from here on references to agent are meant to be for rational agents unless explicitly stated otherwise). There are many variants on what an agent is and does. There are Mobile Agents (e.g. Voyager, Grasshopper), Learning Agents, Autonomous Agents (e.g. robots), Planning Agents, Simulation Agents, Distributed Agents, and many more. All of these agents are attempts at solutions to various problems and they all have different needs and goals.

With a vast array of agents it may be hard to see what every agent could need. We turn again to Russell and Norvig for help in determining a breakdown of the characteristics that go into these various different agents. All agents can be characterized by having at least the three following components:

- Agents need Sensors
  All agents in order to solve a problem need to be able to observe some environment in order to determine the right course of action. This observation is done through *Sensors* which give the agent some state information about it's environment. The information collected can be anything from temperatures in a room to distances from the agent to walls. The sensors themselves can be implemented by hardware such as sonar, radar or thermostats, or in software by reading in a line of text that a user enters.
- Agents need to effect the environment
  In order to solve a problem an agent needs to be able to perform some actions on it's environment. This action can really be anything, including doing nothing. In many applications of agents the actions that are available to the agent are to effect, for example by moving itself to a new location. An action in an environment that is initiated by an agent is called an *effector.*
- There must be a component that selects effects
  In order for an agent to act rationally it must make decisions. The decision making process is encompassed in a component of all agents called a *performance element.* A performance element must make decisions on which effector that it would be best to use.

Additionally agents may have any of the following three components in addition to the above three minimum components:

- Many agents try to learn; for these agents there must be some learning mechanism
  We call this learning mechanism the learning element. For the last 20 years there has been extensive research into the t of many modern agents, and new technologies are being developed constantly *learning elements*. The learning component may be a neural network or a decision tree or any of a number of other types of learning algorithms and schemes.
- If an agent is learning it may want to know how well it is doing
  If an agent is trying to become better at solving a task then occasionally it may want to know if it is doing better or worse than it was doing before so it can make more intelligent decisions on what to change in order to become better. This monitor of performance is called a *critic*.
- Some agents need the capability to explore
  An agent that needs to explore or that desires to explore new solutions or paths would need another component. This component called a *problem generator,* is responsible for suggesting actions that would lead to new experiences.

These six components: sensors, effectors, performance element, learning element, critic, and problem generator used in various different combinations and in different ways can create almost all types of agents, some more easily than others but nearly every agent can be modeled with these components. One broad category of agents however cannot be modeled with these 6 components and that category is distributed agents. Before we can analyze the needs of a distributed agent we must learn what a distributed system is and why it would be useful to agents.

D. Distributed Systems

1) What is a distributed system: Tanenbaum and Steen[10] loosely define a distributed system as, "A collection of independent computers that appears to its users as a single coherent system." There are four general goals of distributed computing they are:

1) connect users to resources
   Connecting users and resources is really the primary goal of distributed computing. There are several aspects that need to be addressed when considering connections between users and resources, the main concern being security.
2) hide the fact that resources are distributed
   Hiding the distribution of resources is called *transparency*. There are various forms of transparency which refer to different ways in which we hide information from users.
3) openness
   To have an open distribution system means to have a system that has a standard way of interfacing with the system so that it becomes easier to exchange resources.
4) scalability
   In distributed computing this is a key concept. We want it to be easy to add more computers to the distributed system without having to do anything so we can grow and shrink the system with our needs.

2) Characteristics of Distributed Systems: Distributed systems can be categorized using the four above properties. There are many design challenges and decisions to be made in developing a distributed system. We outline the major decisions here.

- Client Server vs. Peer-to-Peer

The first determination to make is whether the system will be client-server based, a peer-to-peer system or some hybrid of the two. Most text and current implementations of distributed systems focus on a client server based system, in fact Tanenbaum and Steen do not mention peer-to-peer solutions at all.

There are several benefits and shortcomings of both system. The main benefit of a server based system is that you have one computer that monitors and controls accesses and resources on all other machines. However in a massive system this can be detrimental due to the overwhelming amount of communication with the server causing the whole system to slow to a halt.

- Communication Models

The next design decision we need to make is what communications model is appropriate. There are several to choose from including: Remote Procedure call (RPC), Remote Object Invocations, Message-Oriented Communications, and Stream-Oriented Communication. Each of these has it's own benefits and shortcomings. In RPC a process on machine A makes a call to a procedure that can run on machine B. The caller on machine A halts while machine B runs the procedure, with no message passing visible to the programmer.

RPC accomplishes this by automatically generating stub implementations and an empty interface for the programmer to fill in. when the programmer has completed his code the application works as follows. When the client makes a call to a remote procedure it actually makes a call to the stub implementation which sends the parameters to the stub at the server side and then blocks waiting for a response. The server implementation gets the parameters from it's stub, runs it's procedure, and returns the result to it's stub. The server stub then sends the result back to the client stub which passes the result on to the client. All of this hidden from the programmer.

In an object oriented language we have the ability to create objects and distributed computing takes that encapsulation of objects and uses it in a distributed setting by placing objects on different machines and then hiding the calls from one object to the next. All objects have methods to operate on the object which are defined in interfaces. By having a separation between interface and method we can place the object and all it's methods on one machine and the interface to that object on other machines. When a client binds or connects to a distributed object that object's interface is passed to the client in what is called a proxy. A proxy is similar to an RPC stub, managing communication between the client and the distributed object. As in RPC there is a server side stub as well called a skeleton which handles all the communication at the server side. The calling of remote objects methods is called *remote method invocation*.

There are two situations to consider for message oriented communication: when the receiver is running and when the receiver is not running. There are several message passing paradigms for when receivers are running and there are message queuing paradigms for when the receiver may not be running. The Message Passing Interface (MPI) standard was developed for hardware independent communication which allows developers to easily send and receive messages between machines. This works well for transient communication where if the receiver of a message cannot be reached then the message is dropped. For persistent communication we would need a message queuing model where messages are queued and stored until they can be delivered to the recipient.

Finally Stream oriented communication gives us support for continuous media. This is particularly useful in distributed applications for real-time information such as graphics or sound which strive to update continuously. Most streaming communications conform to the token bucket model where an application sends data to a bucket and the bucket sends out a regularly timed stream of data.

- Transparency

There are eight forms of transparency in distributed computing, each dealing with how to hide distribution of resources from the user:

- Access - Hide how resources are accessed
- Location - Hide where a resource is located
- Migration - Hide the movement of resources
- Relocation - Hide the movement of resources while they are in use
- Replication - Hide the duplication of resources on multiple systems
- Concurrency - Hide the sharing of resources among processes
- Failure - Hide the removal and addition of resources
- Persistence - Hide whether a resource is on disk or in memory

Developing distributed systems requires that one balances the trade-off between performance and transparency. In general the more transparent a system is to the user the worse it performs. So we must be careful in determining which forms of transparency are important to our applications and what forms of transparency can we avoid thus forcing the user to deal with directly.

- Security measures

Determining security levels is also an essential component to all distributed systems. Because distributed systems pass information and resources between machines there is an inherent risk associated with them. Security levels directly effect the openness that distributed systems try to accommodate. Therefore, a balance must be struck between a systems desire to expand and how easily it shares information.

There are many security aspects to consider and we don't hope to cover or explain all security mechanisms here but some of the more important concepts in security are: encryption, firewalls, access control lists, and secure keys. Of course this isn't a complete listing and an explanation security measures inside or outside of distributed computing is outside the scope of this thesis.

E. Distributed Agents

1) What is a Distributed Agent: We define a *distributed agent* as an agent that accepts input selects an action and performs that action while communicating with other agents with some degree of transparency to the developer and to the user. With this definition we are leaving the design decisions of security measures and whether the system is implemented using a client-server or peer-to-peer protocol arbitrarily up to the developer of the system. By having a considerably broad definition of what a distributed agent is and does we leave ourselves with a wide range of possible applications.

2) Characteristics of Distributed Agents: Since distributed agents consist of specific implementations of distributed systems and specific implementations of rational agents we can easily conclude that the characteristics of distributed agents derive from these two categories. Therefore there are many implementation decision in the creation of distributed agents.

In order to implement a distributed agent system one must carefully analyze the goals to which the system is working to achieve. There are several key trade-offs to consider, as well as several more subtle yet still important aspects to consider, all of which are derivations from our discussion in the above two subsections. Below we analyze these trade-offs and how they are related to the goals of distributed agents.

- peer-to-peer or client-server based

This decision has many implications not only for how the system handles communication and hierarchies, but also for many other components in the agents. A peer-to-peer based solution works well for a many-agent system because communication propagates around a network quickly, however it is also much harder to implement because passing messages between two specific agents becomes more difficult and more attention needs to be devoted to load balancing by the developer. Client-server based solutions are much easier to develop but can quickly bottleneck because all traffic must route through the server.

Also this decision affects how messages are passed and what information must be contained in messages. Client-Server solutions generally have a more simple communication protocol because some management of messages can be handled by the server. Whereas peer-to-peer solutions have no means of managing communication other than in the message that is being passed around.

- Distributed objects

In general, distributed objects is the way to go with distributed agents. The object oriented model is well suited for developing agents that can then share interfaces to communicate. However this requires some form of communication agreement between all agents. So the decision then becomes how open does the system need to be. Will there be new agents added to the system regularly. If that is the case then special attention is needed to ensure that the interface standard is easy to use and well thought out, doing everything possible to not inhibit the developers of agents for the system. The Foundation for Intelligent Physical Agents

(FIPA)[11] is attempting to create a standard protocol for agent communication which uses a distributed object framework.

- Transparency

The level of transparency in the distributed system is another key decision. Many agent architectures go to great lengths for distribution transparency, this is evident in the FIPA standard. In developing a distributed agent architecture one needs to evaluate the goals of the system and determine what levels of transparency are appropriate for both the developer of an agent and the end user of the agent system.

- Are the agents going to have learning capabilities

This is an essential determination for all distributed agent architectures. This will determine how complex the agent is. Generally using non-learning agents there is a significant reduction in communication. This is because agents don't change; they change their state or make decisions and they broadcast this information. Also in non-learning systems generally there is a reduction in processing in an agent. If the agent is not learning new things then it generally has less to compute in a given time interval and therefore can respond quicker. If the agents in a system learn then generally they take longer to process and they need to communicate more information, and at the very least they need to communicate what they are capable of learning. All of this is important for figuring out if the system needs a communication language between agents or whether a distributed agent system is necessary at all. This decision further affects what communication model the system uses and whether it's peer-to-peer or client-server based.

The list of above key decisions is far from complete and depending on the system that is being designed other considerations may become more important while these decisions become insignificant, but for the vast majority of scenarios these four decisions are an excellent starting point for deciding how a system should be implemented.

III. Related Work and Methodologies

With the potential advances in the areas of distributed AI it comes as no surprise that there are a plethora of related projects and work. It would be impossible to mention all the different solutions that have been implemented for both the research and commercial communities. After spending time looking through numerous projects and research papers it became clear that many solutions were derived for specific situations. It also became clear that there were several methodologies or standards that were commonly followed. Therefore in this section instead of tackling a list of specific solutions we look at three methodologies that were used in multiple projects.

We analyze these standards and methodologies for several reasons but the most important reason and conclusion that can be drawn during our evaluation of previous work is that currently most developers choose to develop their own original solution, sometimes based on a standard. Therefore we are led to analyze the standards and determine their usefulness for the construction of Loom.

In this section we present three such standards. In the next three sub-sections we look at the Multi-agent System Engineering (MaSE) developed at the Air Force Institute of Technology, the Open Agent Architecture developed at the SRI AI lab, and the Foundation for Intelligent Physical Agents (FIPA) standard.

A. Multiagent System Engineering

The Multiagent System Engineering (MaSE)[12] methodology developed at the Air Force Institute of Technology is a tool for developing distributed agents. It is bound tightly with an implementation of the methodology called AgentTool[13]. This tool gives us a good methodology to implementation mapping enabling us to make a good analysis of the methodology. The MaSE is structured in such a way to help the developer of a distributed agent system to visualize the design process of what he or she may need to do in order to create the agent. Therefore in AgentTool, which gives the user a graphical representation of agents, agent components and communication networks, the developer is given a set of components from which to work. And after assembling components the user has an agent. This is what the MaSE supports - the design process and not as much the design itself. A developer using AgentTool or any other tool on top of the MaSE infrastructure is still required to determine what agents are needed and how to construct them.

B. Open Agent Architecture

As described in the last section there is a tool available called the Open Agent Architecture (OAA) which seeks to develop a methodology for developing distributed agents and there have been several development toolkits that have implemented the OAA. One example of a tool that is implements the OAA is the Agent Development Toolkit (ADT).[14] The ADT provides a variety of mechanisms that support the specification and implementation of individual agents, as well as cooperating communities of agents.

The OAA model seeks to encapsulate six characteristics. These are:

- Open: agents can be created in multiple programming languages and interface with existing legacy systems.
- Extensible: agents can be added or replaced individually at runtime.
- Distributed: agents can be spread across any network-enabled computers.
- Parallel: agents can cooperate or compete on tasks in parallel.
- Mobile: lightweight user interfaces can run on handheld PDA's or in a web browser using Java or HTML and most applications can be run through a telephone-only interface.
- Multimodal: When communicating with agents, handwriting, speech, pen gestures and direct manipulation (GUIs) can be combined in a natural way.

The OAA has many of the same goals as we are striving to accomplish, and many people have acknowledged that the OAA has a good methodology for some distributed agent situations. However there are several pitfalls to the OAA methodology. The primary of these is that it uses a client-server based solution. This significantly limits the scalability of a system developed using the OAA. Also the system does not specify how an agent should be constructed. It leaves a significant amount of freedom to the developers as to how to develop their agents. Which in some circumstances is acceptable but a definitive agent model is necessary if all agents developed could be considered part of the same distributed system since they would all be able to determine what the others were doing.

C. Foundation for Intelligent Physical Agents

The Foundation for Intelligent Physical Agents (FIPA) also hosts a potential standard for developing distributed agents. FIPA has a set of proposed standards and schemata for the development of distributed agents. FIPA itself is purely conceptual in nature and has no specific implementation. There are different standards for each component of the distributed system. By partitioning different components of agents into separate standards the entire set of standards which FIPA has developed has grown out of proportions. There are hundreds of pages detailing the various standards, and although there are some great general models in the FIPA standard, in which Loom uses, they must be abstracted out of the complex detail that is supplied. Thus making the standard nearly unusable to the average developer.

D. conclusions on related work

After reviewing distributed agent systems we saw that there are only a few tools that are capable of providing developers with the a way to develop agents for a distributed setting. And therefore nearly all solutions of distributed agents are focused on the specific problem. Further, we also deduced that many developers of AI in general have no formal training in AI. Particularly in the commercial setting, AI is not the goal it is only the means and thus is not given as much development emphasis as it could be. Therefore a rapid development tool would greatly advance the ability of developers to solve the problems they need to solve without being hampered by also developing an AI system. In the next section we address this concern more and lay out the goals of our system.

IV. Goals of Loom

Rapid prototyping is already a common practice among today's manufactures and many businesses ranging from architecture to software engineering have some form of prototyping in place. In the software industry more reliance is being placed on the computer's ability to make decisions and AI is being pushed forward in the development cycle. The need to be able to produce AI early in the development cycle presents an intriguing problem. In many applications the AI can't be developed until the application is nearly complete, but if the application needs AI early in the development cycle then how do we proceed? The answer is in a pluggable, easy to use development platform.

In this chapter we discuss what we believe the design goals of a successful distributed agent development framework should be. We will not discuss implementation or design decisions until the next chapter; here we are strictly focused on the goals that the Loom team set out to conquer and present explanations of why we believe that these goals are essential to a development platform for distributed agents. It is important to point out that these goals are not for a specific implementation of a distributed agent system, these goals are for designing a distributed agent development framework for developers.

We believe that there are five vital goals to the development framework. The system should have a minimal and simple application programming Interface (API). The system should have pluggable components. The system should provide developers with a maximum level of transparency, while allowing advanced developers the ability to modify elements. The System should be scalable, and be able to accommodate any number of agents. Finally the system should allow for a wide range of possible agent types. We address each of these main goals below.

A. Minimal and simple API

No matter what kind of development tool is being designed the minimal, most simple, and most intuitive API is important, but for an artificial intelligence development platform this is even more important. The goal of Today's applications is not to create agents and artificial intelligence; the applications however do require AI in order to reach their goal. The consequences of AI not being the goal of applications is that there are very few developers that have experience developing AI algorithms, let alone a whole architecture that facilitates AI. Therefore the design of a framework to develop AI and agents must be simple, easy to learn, and as small of an API as possible.

B. Pluggable Components

Because of the diverse range of agents that could be developed and because of the growth in AI in the last 10 years the system should be flexible enough to allow components of agents to be swapped out and replaced quickly and easily. However diversity in agent types is not the reason for having a pluggable architecture. The main reason is because of the dilemma presented above in the introduction to this chapter, where agents are needed early. By having a pluggable architecture we allow ourselves the freedom of quickly prototyping some "dumb" agents in order to test and develop the rest of the system and then when the system is further along in the development process more time can be devoted to creating a "smart" agent. This flexibility in development generates an attractive model for developers that are new to distributed agents or AI in general because they can create simple agents easily and slowly generate better agents as their skill set expands.

C. Maximum Transparency

A goal that we believe is necessary is to hide the details of the distribution as much as possible but at the same time allow access to them so not to restrict the developer to a single implementation. This goal is focused on making the system flexible as well as transparent. We want the details of the distributed communication to be as hidden from the developers as possible. However we want to allow the developers the ability to specify elements if they need to for their purposes. This clashes with our desire for a simple API since the more we open up to the developer the more complex the API becomes. In the development framework a balance must be drawn between these two components.

D. Scalability

Scalability is an important consideration in any distributed system and in a distributed agent system it is no different. A distributed agent development platform isn't too concerned with scalability in and of itself, but with allowing the developer the ability to add as many agents as he or she may wish to add, and also make it possible to add them later. This can be addressed with two different approaches, the first approach is to leave the development framework open thus forcing the developer to make sure that his or her system will scale well ultimately giving more power to the developer. The second approach is to build a framework that inherently supports an open and scalable system but ultimately restricts the developer from controlling certain aspects of how agents are distributed and how they communicate. We believe that in order to develop a simple and easy to use system that the second approach is the appropriate solution. At the same time we try to allow for as much control as we can into the system while keeping the API basic and simple.

E. Allow for as many types of agents as possible

This goal is somewhat obvious but many development frameworks are centered on developing just one kind of agent, or on a subset of agents. We believe that this is unacceptable and don't think that multiple tools should be used for different types of agents. Although a development framework that supported the creation of all types of agents is probably outside the realms of possibility we still set the goal to create such as system in hopes of supporting a maximum amount of agents.

V. Architecture of Loom

The previous chapters have dealt with understanding the obstacles and design considerations that go into the development of a distributed agent system. Having developed this foundation we proceed in this chapter with the architecture of Loom. As explained in the last section there were five main goals in developing the Loom framework:

- simple API
- pluggable components
- high degree of transparency
- scalability
- generalization of agents

These goals as well as the foundation laid out in the previous chapters allow us to decompose the architecture of loom into two convenient components. First we will look at the architecture we choose for an agent in Loom and discuss how it meets our goals. We continue by looking at how loom handles communication and the distributed aspects in the development framework.

A. High Level overview

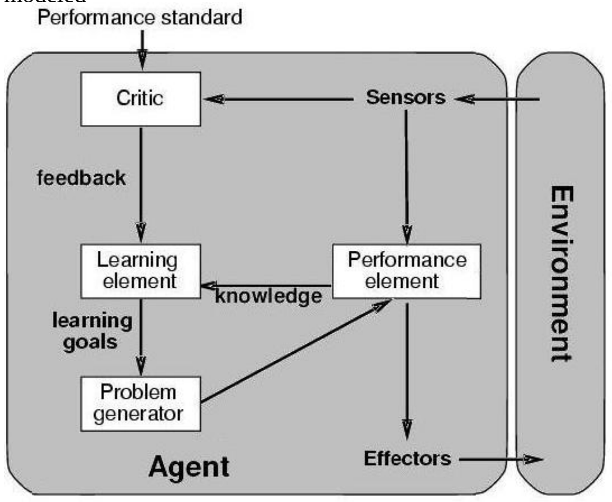
The Loom architecture allows a developer to build agents with a few lines of code. To extend the concept of an agent to encompass more and different types of decision making paradigms as well as new types of effects and senses that agents are capable of. An agent can be any of seven different agent types, or it can be a combination of any number of them. The seven types of agents are:

- REQUESTER - I am waiting for an agent to determine the appropriate action for me.
- SOLVER- I am solving a problem for another agent.
- DRONE - I'm being told what effector to use
- PARENT - Tell a DRONE what to do
- SUBJECT - Every time I use an effector I tell my listener that I'm doing something
- LISTENER - I'm receiving updates from subjects about what they are doing
- STANDALONE - this is the default mode. The agent reads in his sensors and decides what effector he should use.

B. Agent Architecture

We started our design by first looking at how to develop a standalone agent. Out of the five goals for our system three apply directly to a standalone agent: general, simple API and pluggable. After working through several iterations we implemented two prototypes. Both of which solved our goals of a generalized AI and had moderately simple APIs. However the APIs had some redundancies and had a few complex

Fig. 1. A generalized abstraction of how many agents can be modeled



areas and neither system gave us pluggable components. Our last attempt has produced excellent results meeting all three goals. The design was developed based on an agent model presented by Russell and Norvig and repeated in figure 1.

This model gives us incredible flexibility within the agent to create nearly all types of agents. By utilizing the sensors, effectors and performance element in this model we can generate all of the basic, non-learning, agents. By including the learning element we allow the developer the flexibility to plug in a neural network or decision tree or any number of learning based algorithms to his agent.

Although this model is conceptually very well organized, in order to create a clean implementation we deviated from it in several areas. We implemented the agent using a mediator[15]. The mediator pattern gives the control of the agent to the performance element. This was an essential decision for the overall system as well, which is outlined below. Our implementation of an agent is diagrammed in figure 3. This diagram however is missing the the distributed communication components which we will describe in detail later in this chapter.

At the center of the diagram is the agent object, which is the mediator for the object. Every cycle the Agent checks it's sensors, passes on new information to the learner, which is the learning element out of figure 1. The learner may check with it's critic to see how well it's doing. Then returns control to the agent. The problem generator is then queried to see if we should do anything irregular, finally the effector is told which effect to use. This model works well because the agent object maintains control of what the agent is doing at all times. It prohibits us from taking advantage of parallel computing within the agent, however we feel that generally agents will run in serial in nearly all cases and therefore creating a threading implementation within the agent was not a concern for us. The time-line in figure 2 shows us the control flow of an agent.

Our model allows us to expose a very small portion of the agent to the end user through the mediator: the Agent object. The agent object then communicates with all it's components but all the communication starts with the agent object. This design gives us the modular design we wanted, exposes a core API that only directly affects the agent, and a general model of an agent which allows for nearly all types of agents. It satisfies all of our goals for an agent. Next, we introduce the final two components of our system that allow us to create multiple agents and allow them to interact.

The first of these components is a proxy[15] between the agent and the decider. The proxy allows the agent and the decider to continue operating independently in a distributed environment. The ability to act independently is

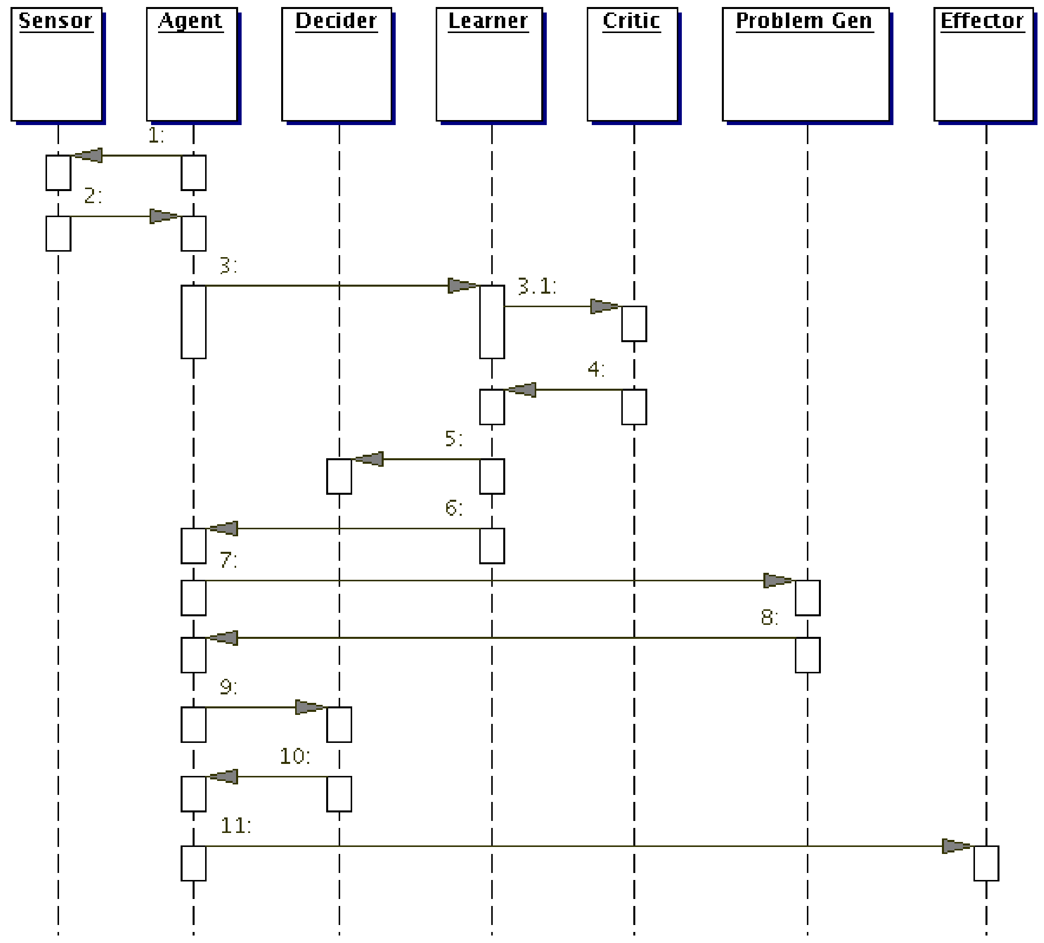


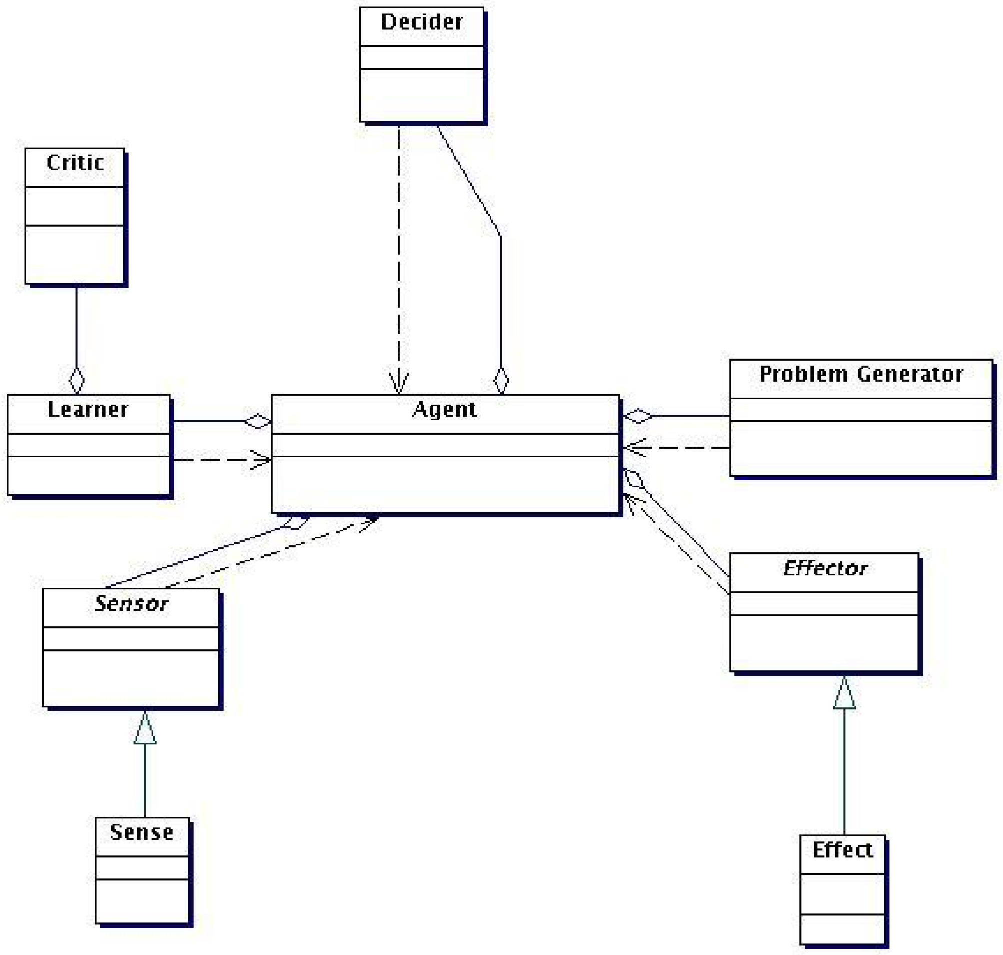
Fig. 2. execution cycle for an agent, shown with all the standalone components.

necessary in order to keep a modular design while allowing the distributed communication to be channeled to the appropriate destinations. The second component is the agent collector which allows multiple agents to talk to one another as well as manage all agents on a given node of the Loom network. We explain both of these components in detail in the next two subsections.

C. The AgentCollector

We wanted to allow for multiple agents at each node within a loom network. This required us to create a platform at each node which handled the communication between the network and all the agents that reside on that machine. In Loom we call this platform the AgentCollector. The AgentCollector handles the registration and initialization of each agent. Therefore, because the initialization is handled at the AgentCollector, it is required even for a system with only one agent.

Fig. 3. This is the basic agent architecture.



During the initialization of an agent the AgentCollector gives the agent a universally unique identifier (UUID)[16] which is used internally to Loom to identify each agent. Then the AgentCollector spawns a thread for the agent and starts the agent running in that thread. The agent then begins its execution cycle as shown in figure 2 until it is again interrupted by the AgentCollector or until it is unregistered from the AgentCollector.

The AgentCollector is also responsible for all communication between agents. When an agent broadcasts a request or when an agent is communicating directly with another agent all communication is sent from that agent to the AgentCollector. The AgentCollector is responsible for packaging the information from the agent into a packet that can be sent across the network. It is also responsible for unpacking received packets, determining if the message is for any of its agents and delivering the message to the appropriate agent.

D. Agent-Decider proxy

In order for agents to communicate between one another we needed some way of telling an agent to do two things that normally it would handle itself. We needed to tell the agent to not listen to its own sensors but to use some other copy of them that would be supplied, and we needed to tell an agent that instead of performing the effector that the decider returned to instead send that effector to another agent. This was a hard problem to overcome because the flow of execution through an agent was originally very clean. The solution was in creating a proxy between the agent and the decider.

As explained before an agent can be in any combination of the seven agent types. When an agent type is set the proxy determines where the sensors will be coming from and where the effector will be going to. When the agent is in STANDALONE mode the proxy passes the agents own sensors in to the decider and when the decider returns, the proxy returns the effector to the agent. However in a distributed mode an agent A may ask an agent B to calculate an effector based on specific sensor input. In this scenario agent B sends agent A a set of sensors. The proxy in agent A passes these sensors results into the decider instead of agent A's. When agent A's decider returns the proxy doesn't pass the result to agent A, instead he passes it on to the AgentCollector as a return message. If agent A is in two modes at the same time then the effectors may be sent to multiple requesting agents at the same time. With this proxy solution an agent can now handle the distributed communication that Loom demands.

E. Full System Architecture

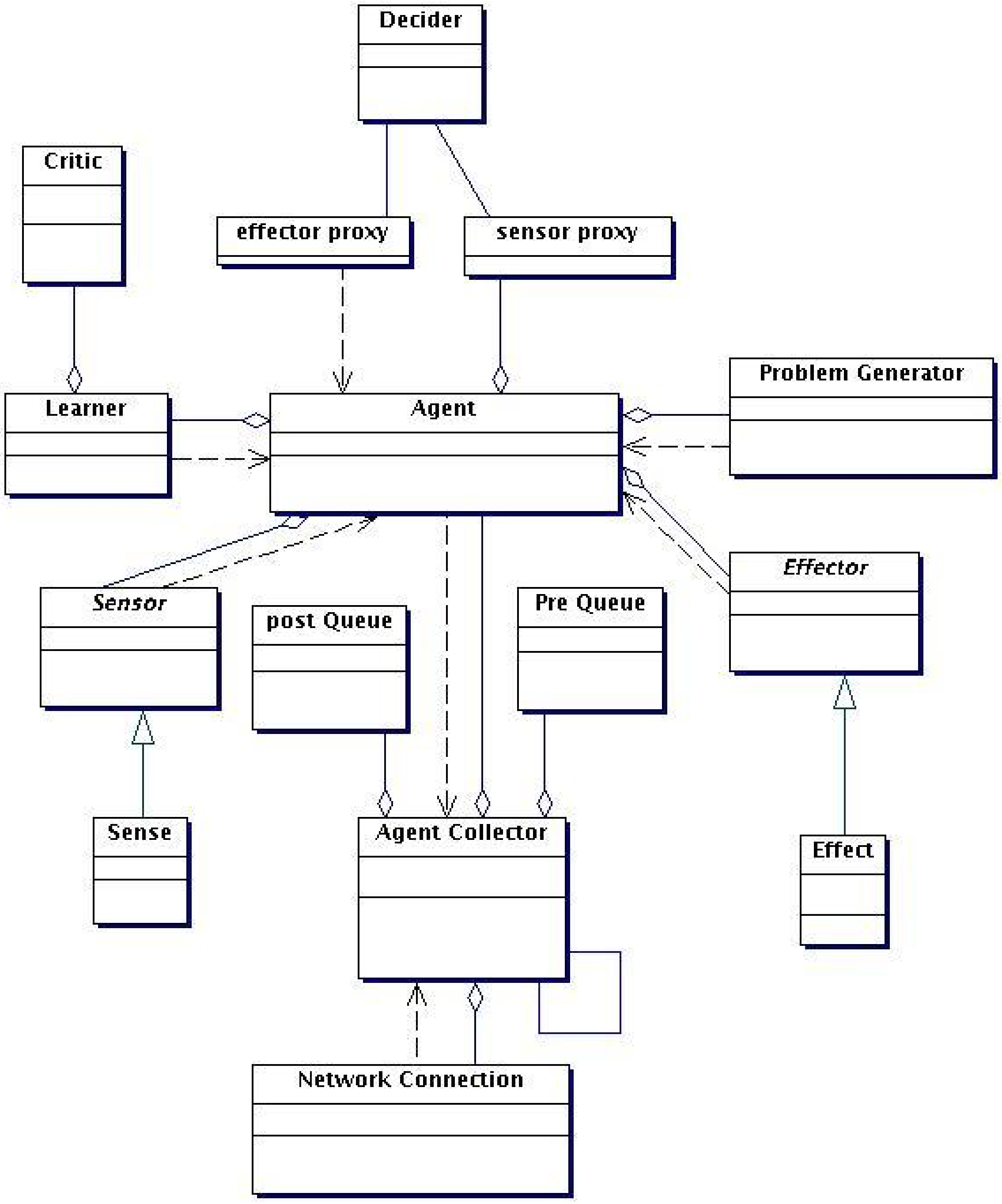
Our system is now nearly complete and we present a complete model in figure 4.

You will notice that there are three components in this representation that we have not yet discussed. The AgentCollector has a post-queue, a pre-queue and a Network Connection. These three components are part of the AgentCollector and help with the handling of distributed communications. The network architecture and the handling of distributed communication is presented in the next subsection and is the final component of Loom's architecture.

F. Network Architecture

The AgentCollector's architecture has undergone an iterative design cycle. The first implementation that was proposed was a non-distributed hierarchically based in which there were low-level agents that had reactions to their environments and passed those reactions up to more abstract agents who in turn may pass their reactions to more abstract agents. This implementation was a good starting point but had many drawbacks, specifically it limited the developer to creating agents that only could do certain tasks either reactions or decision making or knowledge implications or any one specific tasks. The second implementation that we developed was a non-distributed model very similar to the current architecture with agents having various components that were decoupled but were not configurable. We realized that a distributed solution was a key factor in allowing developers the freedom to develop current systems. As presented in the first chapters of this thesis, there are numerous reasons for this decision and therefore we came to our current implementation.

Fig. 4. An agent and its relationship with other Loom components



The AgentCollector architecture was originally developed for a non-distributed solution and has since been modified for a peer-to-peer based solution. We believe that the current and future demands on distributed agents will grow significantly and that a large population of agents will be necessary to complete tasks in every-day scenarios, thus leading to the peer-to-peer solution. Because the AgentCollector is a peer-to-peer implementation it does not need significant restructuring from it's original form and an abstract communication layer can be added without sever repercussions to the rest of the system.

For the communications layer we use a library called plexus. Plexus handles all the peer-to-peer message passing. It uses a reliable communication protocol and handles all the routing of messages throughout the network. To use the plexus networking functionality we wrap the creation, connection, sending, and receiving functionality into our own object which we call a NetworkConnection and can be seen at the bottom of figure 4. This singular facade supplied to the network greatly increases the simplicity of the system to the developer. However this facade means that after a message is decoded by an AgentCollector coming off the plexus network that it must be propagated down into the decider. In figure 5 we can see that when a message comes in to the AgentCollector from the NetworkConnection it determines if the message is for any agent that it has. If it does have the appropriate agent then it puts it in one of the queues. based on the type of message that is received. If the message is to perform an effector then it goes in the postQueue if it is to use a specific sensor then the message goes in the pre-Queue.

When the agent updates each frame it checks the queues and pops off any messages waiting for it. The messages are then passed to the agents appropriate proxy. The decider makes his decision and passes it to the effector proxy which then performs the correct operation based on what type of agent it is. If the agent is a SUBJECT then the effector proxy generates a message to send to the LISTENER and sends it to the AgentCollector who gives it to the NetworkConnection and then plexus broadcasts the message out.

G. *Loom Architecture conclusions*

The Loom architecture has been iterated on several times. The current version of Loom has many advantages over it's predecessors. The current design meets the goals of having a simple API, generalized model of an agent, a modular component approach, transparent communication and scalability. Although the internal components of Loom have a complex interaction scheme, the external API is simple and easy to understand. We believe that the architecture of Loom allows a developer with no agent design experience a great framework to begin with.

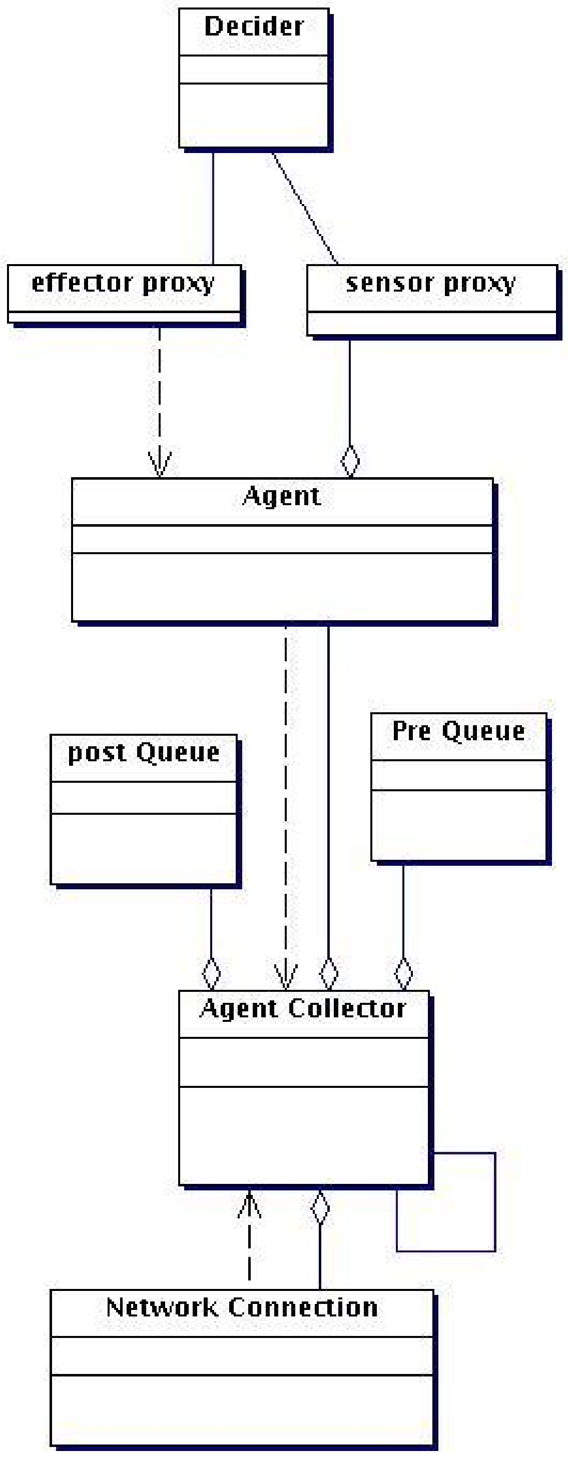
VI. Evaluation Plan

Our evaluation plan is fairly simple. Since the design goals of Loom are mostly centered around simplicity and ease of development our primary evaluation will be to determine if the system is actually usable and easy to learn. We also want to test the reliability and the performance of the system so we also have a plan for testing both of these components. All three testable components are listed in this section.

A. *Usability*

We plan to implement at least 2 separate distributed systems using loom upon it's com-

Fig. 5. All the components involved in propagating a message in Loom.



pletion. One system will be a test application that will be packaged with loom and thus will be used as an example, it will contain several agents and communicate with different messages. The second package will contain just two agents that communicate back and forth, creating a good simple test case for the entirety of the system. Using a unit testing analogy we will test how easy it is to develop each component of the system and how intuitive it is to understand quickly. Of course this is a biased test since the developer is testing the usability of the system therefore if time permits I will ask someone else to develop a simple application using the system and ask them to give feedback on the usability of the system.

B. *Performance*

We will measure performance using the first test system that we implement. We will test the latency in message sending and receiving with a varying amounts of agents in the system. We will also test the latency of end-to-end connections. This is the amount of time it takes for a message in the peer-to-peer network to be received by an AgentCollector determine that it is not for his agents and continue passing the message on.

C. *Reliability*

Because we currently implement the communication in a non-reliable manner, we will also test the reliability of the system over a non local network to see what kind of results we obtain. Then using these results we will determine if it is necessary to implement a reliable communications protocol in the next iteration of Loom's development.

VII. Conclusions

After developing on Loom for two months the system now supports half the agent types. Currently the supported agent types are STANDALONE, DRONE and PARENT; in a semester we were only able to complete work on these three types. In this section we address the five goals that loom set out to accomplish based on the three supported agent types.

A. *minimal and simple API*

We are very satisfied with the result of the external API to the Loom library. In order to have an intelligent agent we only needed to write three methods (each approximately ten lines long). In order to write a distributed loom application which would join a loom network and start sending out messages it only took a nine line application. The exposed API is less than 20 methods for the entire system. When given to a developer for the first time he was able to learn loom and write an agent in less than 1 hour.

B. *pluggable components*

Nearly every element in an agent is pluggable. At run-time it is possible to change sensors, effectors, and deciders. We can also change the agent's type and what agents it is communicating with. This services our goal of pluggable components as the developer has full access into the agent to modify it as he or she sees fit.

C. *Maximum transparency*

Because of Looms architecture the developer is shielded from the distributed components nearly entirely. There are only two places the developer is exposed to distributed portion of

Loom. The first is when they implement their own effectors, but even then it is not a direct access to the distributed aspect because they only need to understand the search methods to locate an agent. The routing of messages and the determination of what effects and senses to use are all handled by loom based on what the agent type is. The second aspect of the distribution they need to know is when they are connecting their agent to other agents on the peer-to-peer network then they need to know the port and hostname of the peer. Loom does a good job of hiding the network layer from developers letting them work on the AI and not the distributed component.

D. scalability

Our approach to Loom in using a peer-to-peer methodology has made loom very scalable. The plexus network allows us unlimited expandability, allowing loom networks to grow to as many agents as one might need. We do not have the resources to experiment with such a massive network so it is not possible to comment now on how the system responds to having many agents communicating.

E. Allow for as many agent types as possible

Currently with only three of seven types implemented it cannot be said how many types of agents Loom supports. However because all the components of a standalone agent are in place. we do have the ability to create all of the agent types that have been mentioned in this paper. From look-up agents, to first order logic agents, to machine learning agents the pluggable nature of Loom allows all of these any many more.

VIII. Future Work

There are three milestones yet to complete on Loom. In this section we briefly outline what each of those milestones is and how we believe we can attain them.

A. Complete remaining four agent types

Now that the communications protocol and message propagation is in place we need to finish our work on the various agent types. There is one major obstacle that we will need to resolve before being able to complete this section; we will need to be able to send more then just effector names and sensor names over the plexus network. The messages that we generate will need to be able to package lists of these types as well as generic message types and searches. Once the message object structure is revised to be able to handle these new types then adding additional agent types will be a straight-foreword process.

B. Create a graphical editor for building Agents

This graphical editor would allow a developer to visualize the agents they are building and help them further determine what portions they still need to implement. This would be done in much the same way as Microsoft's Visual Basic component object model architecture works; when a developer adds a component it prompts the developer for code where it is needed. This would add an invaluable amount of clarity to the development of agents and would help those developers that are not familiar with AI in visualizing what the agent is doing.

C. Scalability and performance testing

Once more agent types are added to Loom we will be able to test more accurately how the system responds in a many-agent system with more than 4 AgentCollectors. Once we have a large network set up we can more accurately test how well the networking component works as well as how well the agents perform with the addition of more communication.

References

[1] "Amazon website."

[2] "Roombavac website."

[3] "openai website."

[4] "Open agent architecture website," 2003.

[5] S. Russell and P. Norvig, *Aritifical Intelligence - A modern Approach*. New Jersey, NJ: Prentice Hall Artificial Intelligence Series, 1995.

[6] J. Haugeland, ed., *Artificial Intelligence: The Very Idea*. Cambridge, MA: MIT Press, 1985.

[7] R. Kurzweil, *The Age of Intelligent Machines*. Cambridge, MA: MIT Press, 1990.

[8] E. Charniak and D. McDermott, *Inroduction to Artificial Intelligence*. Potomac, MA: Addison-Wesley publishing, 1987.

[9] R. J. Schalkoff, *Aritificial Intelligence: An Engineering Approach*. New York, NY: McGraw-Hill, 1990.

[10] A. S. Tannenbaum and M. van Steen, *Distributed Systems - Principles and Paradigms*. New Jersey, NJ: Prentice Hall, 2002.

[11] "Foundation for intelligent physical agents website," 2003.

[12] M. F. W. Scott A. DeLoach and C. H. Sparkman, "Multi systems enginnering," *Internation Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 3, 2001.

[13] S. A. DeLoach and M. Wood, "Developing multiagent systems with agenttool," 2000.

[14] D. L. Martin, A. Cheyer, and G. L. Lee, "Agent development tools for the open agent architecture," in *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, (London), pp. 387–404, The Practical Application Company Ltd., Apr 1996.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, New York, NY: Addison-Wesley Publishing Company, 1995.

[16] M. Mealling, P. J. Leach, and R. Salz, "A UUID URN namespace," tech. rep., Internet Engineering Task Force, 2002.