ADAPTIVE CLONING APPROACH TO AGENT MOBILITY

A DISSERTATION

submitted in partial fulfilment of the requirements for the award of the degree of

MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND TECHNOLOGY

UNIVERSITY OF ROORKEE CENTRAL LIBRARY
Acc. No. 247866
Date 18.3.99
ROORKEE

By

MAHENDAR ALETI

DEPARTMENT OF ELECTRONICS AND COMPUTER ENGINEERING
UNIVERSITY OF ROORKEE
ROORKEE-247 667 (INDIA)

JANUARY, 1999

CANDIDATE'S DECLARATION

I here by declare that the work which is being presented in this dissertation entitled **"ADAPTIVE CLONING APPROACH TO AGENT MOBILITY"**, in partial fulfilment of the requirement for the award of the degree of **Master of Technology** (M.Tech.) in Computer Science and Technology (C.S.T) in the department of Electronics and Computer Engineering, University of Roorkee, Roorkee, is an authentic record of my own work carried out by me for a period of six months form August 1998 to January 1999 under the supervision and guidance of **Dr. MOHAN LAL,** *Asst. Professor, New Computational Facility, and* **Dr A.K. SARJE,** *Professor, Department of Electronics and Computer Engineering, University of Roorkee, Roorkee.*

The matter embodied in this dissertation has not been submitted by me for the award of any other degree or diploma.

Date: 16-1-99

Place: Roorkee **(MAHENDAR ALETI)**

CERTIFICATE

This is to certify that the above statements made by the candidate are correct to the best of our knowledge and belief.

16/1/99

(Dr. MOHAN LAL)
Asst. Professor,
New Computational
Facility,
University of Roorkee,
Roorkee- 247667
Date:
Place: Roorkee

16/1/99

(Dr.A.K.SARJE)
Professor
Dept. of Electronics, &
Computer Engineering
University of Roorkee
Roorkee - 247667
Date:
Place: Roorkee

ACKNOWLEDGEMENTS

It is my proud privilege to express my profound of gratitude to my guide **Dr. MOHAN LAL,** *Assistant Professor, New Computational Facility,* **Dr. A.K. SARJE,** *Professor, Dept. of Electronics & Computer Engineering,* for their invaluable inspiration, guidance and continuous encouragement throughout this dissertation work.

I express my sincere thanks to **Dr. R. C. Joshi,** *(Head of the Dept. of Electronics & Computer Engineering, and Co-ordinator, NCF)* and other staff members for providing necessary facilities for successful completion of this work.

I would like to thank all of my friends for their help and constructive criticism during development of software.

I am very much indebted to my parents for their moral support and encouragement to achieve higher goals.

(MAHENDAR ALETI)

ABSTRACT

Multi Agent Systems provide efficient solutions for several computational problems. Multi Agents systems are subject to performance bottlenecks in cases where agents cannot perform tasks by themselves due to insufficient resources. Solutions to such problems include passing tasks to others or agent migration to remote hosts. But, in this work, agent cloning used as an approach to the problem of local agent overloads. Agent cloning is the action of creating and activating a clone agent (locally or remotely) to perform some or all of an agent's tasks.

Agent cloning subsumes the task transfer and agent mobility. Agent migration can be implemented by creating a clone on a remote machine, transferring the tasks from the original agent to the clone, and dying. Thus agent mobility is an instance of agent cloning. The requirements of implementing a cloning mechanism and its benefits have been studied. The reasoning, decision-making, and actions necessary for an agent with in the system to perform cloning also have been studied. The merging of two agents or self-extinction of underutilized agents is also discussed. For large number of tasks, cloning significantly increases the portion of tasks performed by Multi Agent System. The program is written in Java under Linux operating system.

CONTENTS

CHAPTER 1

INTRODUCTION

An agent is a computational entity which[1]: -

- acts on behalf of other entities in an autonomous fashion
- performs its actions with some level of proactivity and/or reactiveness
- Exhibits some level of the key attributes of learning, co-operation and mobility.

The number and type of application domains [1][2][3] in which agent technologies are being applied to include workflow management, network management, air-traffic control, business process re-engineering, data mining, information retrieval/management, electronic commerce, education, personal digital assistants (PDAs), scheduling/diary management, etc.

Multi Agent Systems:

Multi Agent System (MAS) can be defined[1] as " a loosely coupled network of problem solvers that work together to solve problems that are beyond their individual capabilities". The members of the system may each receive tasks, and perform or delegate for performance by others. Multi agent systems are subject to performance bottlenecks in cases where agents cannot perform tasks by themselves due to insufficient resources. In Multi agents systems, distributed nature of the agents may contribute to the ability to overcome overloading bottlenecks. MAS is a framework which receives a stream of tasks, and agent in MAS assumed to be

autonomous, self-aware, intelligent, and pro-active computational entity. A task is either an executable code or a goal represented at a higher level abstraction.

An agent provides services by performing tasks which either it generates by itself, or other agent delegate to it. If task represented as a higher level abstraction, an agent should be able to transform the abstract goal into more concrete tasks. Concrete tasks examined by an agent to verify whether their performance is within its capabilities. If they are, the agent either performs tasks as given or further decomposes to subtasks, otherwise, the tasks may be delegated to the appropriate agents.

The agents have two important qualities:

- Capabilities, which indicate the types of tasks they can perform,
- Capacities, which indicate the amount of resources the agents, can access for task execution.

The problem studied in this work concern to the situation where the task flow to an agent overloads it. Such overloads are two different general categories:

- An agent in an MAS is overloaded, but the MAS as a whole has the required capabilities and capacities
- The MAS as a whole is overloaded, that is, the agents which make up the MAS do not have the necessary capacities (however, there may be idle resources in the computational system where the agents situated)

As a result of such overloads, the MAS will not perform all of the tasks in time, although the required resources may be available to it.

The solutions are:

- First case- Overloaded agents should pass tasks to other agents, which have the capabilities and capacities to perform them
- Second case- Overloaded agents create new agents to perform excess tasks and utilize unused resources or migrate to other hosts

In this work, "agent cloning"[4] is used as a means for implementing these solutions and studied the reasoning, decision-making, and actions necessary for an agent within the system to perform cloning. Agent cloning is the action of creating and activating a clone agent (locally or remotely) to perform some or all of an agent's tasks.

1.1 Statement of the Problem:

Objectives:

- To analyze the circumstances under which agents should consider cloning
- To study how cloning affects the performance of an MAS.
- To study merging of two agents

1.2 Organization of the Dissertation:

Chapter 1: Introduces Agents, Multi Agent Systems, agent loading, and cloning mechanism

Chapter 2: Deals with the concepts of agents, agent environment, and agent infrastructure requirements. And it also discusses the design paradigms and mobile code technologies.

Chapter 3: Discuss the cloning approach, and requirements for implementing a cloning mechanism. And also discusses the reasoning, decision making, and actions necessary for an agent with in the system to perform cloning. It also discusses the optimal time for cloning mechanism, the merging of two agents, and merging approach used in this work.

Chapter 4: Explains the simulation model of an agent and its environment. Design and implementation of software is also discussed

Chapter 5: Discusses results and gives suggestion for further work

Appendix A: Cloning Pseudo-Code

Appendix B: Merging Pseudo-Code

Appendix C: Back-Propagation Algorithm

Appendix D: Lists the source code

CHAPTER 2

AGENTS

2.1 Introduction to Agents:

With the recent explosive development of computer networking and the Internet, a gap has developed between the sheer amount of information that is available and the ability to process or even locates the interesting pieces. The increased number of available services has also lead to a profusion of mutually incompatible user interfaces, making it difficult for people to actually take advantage of all that is offered. Agents may show a possible way out of this dilemma program that help their users perform routine chores and assist them during complex tasks. In a computer network, mobile agents may move around on behalf of their users, seeking out, filtering and forwarding information or even doing business in their name.

Agents, intelligent agents and agent-based systems have attracted considerable interest from many fields of computer science, most notably artificial intelligence, distributed systems, computer communications and software engineering. Unfortunately there is little consensus among researchers about exactly what they consider an agent to be. An agent can be defined in the general sense of 'anybody who acts on behalf or in the interest of somebody else'.

We need software agents because [1]: -

- more and more everyday tasks are computer-based
- the world is in a midst of an information revolution, resulting in vast

amounts of dynamic and unstructured information

- increasingly more users are untrained
- And therefore users require agents to assist them in order to understand the technically complex world we are in the process of creating.

***Agents have the following attributes* [5]:**

- Autonomy (acts independently)
- Continuity (persists over time)
- Intelligence (can reason)
- Mobility (across machine boundaries)
- Personality (possesses a human-like persona)
- Adaptability (can learn)
- Knowledge (about some domain)
- Conversation (is directed at a high level)
- Authority (has the rights of its human sponsor)
- Collaboration (interacts with other agents and people)

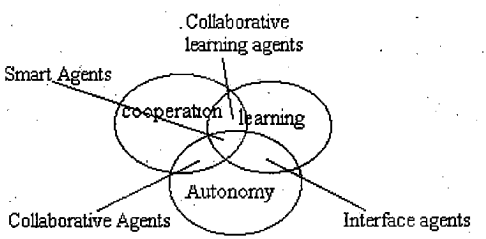

Figure 2.1 shows one agent taxonomy [5], that involve some of the above attributes

***Desirable Agent Characteristics* [6]:**

- Taskable: able to take the direction from humans or other agents
- Network-centric: distributed and self-organizing. When situations warrant, agent mobility might also be desirable
- Semiautonomous: not always under direct human control. For example, in information gathering task, because of the many potential requests for information, humans would be swamped if they had to initiate every information request. The user should be able to control the amount of agent autonomy.
- Persistent: capable of long periods of unattended operation
- Trustworthy: able to reliably serve users needs so that users will develop trust in the agents performance
- Anticipatory: able to anticipate user information needs so that users will develop trust in the agents performance
- Active: able to initiate problem solving activities (for example, by monitoring the Infosphere for the occurrence of given patterns), anticipate user information needs, and bring to the attention of users situation-appropriate information, which involves deciding when to fuse information or present "raw" information
- Collaborative: able to interact with humans and other machine agents. Collaborative interactions allow agents to resolve conflicts and inconsistencies in information, current tasks, and world models, thus improving their decision-support capabilities

- Flexible: able to deal with heterogeneity of other agents and information resources
- Adaptive: able to accommodate changing user needs and task environments

Multi-Agent Systems:

Multi-Agent System, can be defined as "a loosely-coupled network of problem solvers that work together to solve problems that are beyond their individual capabilities"

The motivation for the increasing interest in MAS[1] includes their ability: -

- To solve problems that are too large for a centralized single agent to do due to resource limitations or the sheer risk of having one centralized system;
- To allow for the interconnecting and interoperation of multiple existing legacy systems, e.g., expert systems, decision support systems, etc.;
- To provide solutions to inherently distributed problems, e.g., air traffic control
- To provide solutions which draw from distributed information sources;
- To provide solutions where the expertise is distributed, e.g., in health care provisioning;
- To enhance speed (if communication is kept minimal), reliability (capability to recover from the failure of individual components, with graceful degradation in performance), extensibility (capability to alter the number of processors applied to a problem), the ability to tolerate

uncertain data and knowledge

- To offer conceptual clarity and simplicity of design.

In a broad sense, an agent is any program that acts on behalf of a (human) user. Mobile agent then is a program that represents a user in a computer network and can migrate autonomously from node to node, to perform some computation on behalf of the user. Its tasks, which are determined by the agent application, can range from online shopping to real time device control to distributed computing. Applications can inject mobile agents into a network, either on a predetermined path or one that the agents themselves determine based on dynamically gathered information. Having accomplished their goals, the agents can return to their home site to report their results to the user

2.2 System-Level Issues:

A mobile agent system[7][8] is an infrastructure that implements the agent paradigm. Each machine that intends to host mobile agents must provide a protected agent execution environment. Such agent server execute agent code and provide primitive operations to agent programmers, such as those that allow agents to migrate, communicate, or access host resources. A logical network of servers implements the mobile agent system. Many useful agent applications will require Internet wide access to resources. Because users will need to dispatch agents from laptops, regardless of their physical location, the mechanisms used in the agent infrastructure should scale up to the wide area networks. Agents can execute on many different hosts during their lifetimes.

Agent Mobility:

A mobile agent's primarily identifying characteristic is its ability to autonomously migrate from host to host. Thus, support for agent mobility is a fundamental requirement of the agent infrastructure. An agent can request that its host server transport it to some remote destination. The agent server must then deactivate the agent, capture its state, and transmit it to the server at the remote host. The destination server must restore the agent state and reactivate it at the remote host, thus completing the migration.

An agent state includes all its data, as well as the execution state of its thread, which, at the lowest level, is represented by its execution context and call stack. If this can be captured and transmitted along with the agent, the destination server can reactivate the thread at the precisely the point where migration was initiated, which can be useful for transparent load balancing or fault tolerant programs. Capturing execution states at a higher level, in terms of application defined agent data, offers an alternative. The agent code then can direct the control flow appropriately when the state is restored at the appropriate destination. However this approach only captures execution state at a coarse granularity[8] (such as function level), in contrast to instruction level.

Most agent systems execute agents using commonly available virtual machines or language environments, which usually do not provide thread-level state capture. The agent system developer could modify these virtual machines for this purpose, but such modification renders the system incompatible with standard installations of those virtual machines. Because mobile agents are autonomous,

migration occurs only under explicit programmer control; thus state capture at arbitrary points is usually unnecessary. Most current systems therefore rely on coarse-grained execution state capture to maintain portability

Another issue in implementing agent mobility is the transfer of agent code. In one approach, the agent carries all its code as it migrates, which lets it run on any server that can execute the code. Some systems do not transfer any code at all and require that the agent's code be preinstalled on the destination server. In a third approach, the agent does not carry any code but contains reference to its code base- a server that provides code on request. During the agent's execution, if it needs to use some code not already installed on the agent's current server, the server can contact the code base and download the required code. This is some times called as code on demand.

Naming:

Various entities in the system- such as agents, agent servers, resources, and users -need names that uniquely identify them. An agent should be uniquely named, so that its owner can communicate with or control it while it travels on its itinerary. Having location transparent names at the application level is desirable and can take two forms. The first approach provides local proxies for remote entities, which encapsulate their current location. The system updates this location information when the entity moves, thus providing location transparency at the application level. The alternative approach uses global, location-independent names that do not change when the entity relocates. This approach requires the provision of a name service, which maps a symbolic name to the current location of the named entity.

Security Issues:

The introduction of mobile code in to a network raises several security[8][9] issues. The security-related requirements fall into these categories:

- Agent privacy and integrity;
- Agent and server authentication;
- Authorization and access control; and
- Meeting, charging, and payment mechanisms.

Privacy Integrity:

Agents carry their own code and data as they traverse the network. Parts of their state might be sensitive and might need to be kept secret when they travel on the network. The agent transport protocol needs to provide privacy, to prevent eavesdropper from acquiring sensitive information. Also, an agent might not trust all servers equally. It needs a mechanism to selectively reveal different portions of the agent state to different servers.

Authentication:

When an agent attempts to transport it self to a remote server, the server needs to ascertain the identity of the agent's owner, so that it can decide what rights and privileges to grant the agent in the server environment.

Authorization and access control:

Authorization is the granting of specific resource-access rights to specific principals (such as owners of agents). Because some principals are more trusted than others are, their agents can be granted less restrictive access. For this, resource owners must specify policies for granting access to their resources, based either on

identities of principals, their roles in an organization, or their security classifications. A user might place additional restrictions on her agents' code, so as to limit the damage caused by buggy code. These restrictions can be encoded into the agent's state and enforced by the server.

Metering and charging mechanisms:

When agents travel on a network, they consume resources such as CPU time and disk space at different servers, which might legitimately expect monetary reimbursement for providing such resources. Also, agents might access value-added service or information provided by other agents, which could also expect payment. In market place, users can send agents to conduct purchases on their behalf. Thus, mechanisms must be available so that an agent can carry digital cash and use it to pay for resources it uses.

2.3 Language-Level Issues:

The language level issues [7][8] fall into two categories:

- Agent programming languages and models
- Programming primitives.

Agent Programming Languages and models:

The portability of agent code is a prime requirement, because an agent might execute on heterogeneous machines with varying operating system environments. Therefore, most agent systems are based on interpreted programming languages that provide portable virtual machines for executing agent code. Safety is another important criterion in selecting an agent language. Languages that support type

checking, encapsulation, and restricted memory access are particularly suitable for implementing protected servers.

Several systems use scripting languages such as Tcl, Python, and Perl for coding agents. These allow rapid prototyping for small to medium-size agent programs. However, because script programs often suffer from poor modularization, encapsulation, and performance, some agent systems use object-oriented languages such as Java, Telescript, Or Obliq. These system define agents as a first class objects that encapsulate their state as well as code, while the system supports object migration in the networks. Such systems offer the natural advantage of object oriented in building agent-based applications, complex agent programs are easier to wrote and maintain using object oriented languages. A few systems have also used interpreted versions of traditional procedural languages such as C for agent programming.

Mobile agent systems can defer significantly in the programming model used for developing agents. In some cases, the agent program is merely a script, often with little or no flow control. In others, the script languages borrows features from object oriented programming and extensively supports procedural flow control. Some system model the agent-based application as a set of distributed interacting objects, each having its own thread of control and thus able to migrate autonomously across the network. Other user a call back based programming model in which the system signals certain events at different times in the agent's life cycle. The agent then is programmed as a set of event handling procedures.

Programming primitives:

Agent programming primitives can be categorized into

- Basic agent management: creation, dispatching, cloning, and migration
- Agent-to agent communication and synchronization
- Agent monitoring and control: status queries, and recall and termination of agents
- Fault tolerance: check pointing, exception handling and audit trials
- Security related: encryption, signing, and data signaling

Basic agent management primitives:

An agent-creation primitive [3][7]8] allows the programmer to create instances of agents, there by partitioning the application task among its roving components. This also introduces concurrency into the system. This could be a single procedure to be evaluated remotely, a script, or a language-level object. In object oriented systems, programmers usually create an agent by instantiating a class that provided an agent abstraction. The system can inspect the submitted code to ensure that it conforms to the relevant protocols and doesn't violate security policy. Based on agent creator's identity, the system might also generate a set of credentials for the agent this time. These are transmitted as part of the agent, to allow other entities to identify it unambiguously.

A newly created agent is just passive code, because it has not yet been assigned a thread to execute it. For activation, it must be dispatched to a specific agent's server. The server authenticates the incoming agent using its credentials and determines the privileges to grant it. It then assigns a thread to execute the agent code.

Variant of creation primitive, "agent cloning", allows an agent to create the identical copies of itself, which can execute in parallel with it and potentially visit other hosts performing the same task as their creator. Agent forking, in which the newly created agent retains a parent-child relationship with its creator, is another variant that lets programmers create agents that inherit their ownership and privileges for their parents.

During an agent program's execution, it might determine that it needs to visit another site on the network. To achieve this, it invokes a migration primitive. The destination specified by the agent can be either absolute or relative.

Agent communication and synchronization primitives:

To accomplish useful work, agents often must communicate or synchronize with each other. Systems often use varying mechanisms for establishing inter agent communication. One approach is to provide message–passing primitives, which allow agents to either send asynchronous datagram-style messages or setup stream-based connections to each other.

Method invocation is another approach communication in object based systems. If two agent objects are collocated of a server, they can be provided references to each other, which they use to invoke each other. For agents that are not collocated, the system can provide remote methods invocation.

Collective communication primitives can be useful in applications that use groups of agent for collaborative tasks. Such primitives can provide for communicating with or within an agent group.

Communication can also be implemented by using shared data. Another

metaphor for agent communication is event signaling. Events are usually implemented as asynchronous messages.

Agent monitoring and control primitives:

An agent's parent application might need to monitor the agent's status while it executes on a remote host. If exceptions or errors occur during the agent's execution, the application might need to terminate the agent, which involves tracking the agent's current location and requesting its host server to kill it.

Primitives for Fault Tolerance:

A checkpoint primitive creates a representation of the agent's state that can reside in nonvolatile memory. If an agent (or its host node/server) crashes, the owner can initiate recovery, which can determine the agent's last-known checkpoint and request the server to restart the agent from that state. In addition to the checkpoints themselves, agent servers can also maintain an audit trial to let the owner trace the agent's progress determine the cause of the crash.

Security related problems:

Because agents might pass through untrusted hosts or networks, the agent programmer needs primitive operations for protecting sensitive data. This includes primitives for encryption and decryption that protects the privacy of data, as well as message sealing or message digests that will detect any tampering of the code or data.

2.4 Design Paradigms:

The goal of design is the creation of a software architecture, which can be defined as the decomposition of a software system in terms of software components

and interactions among them. Software architectures with similar characteristics can be represented by architectural styles or deign paradigms, which define architectural abstractions and reference structures that may be instantiated into actual software architectures. A design paradigm is not necessarily induced by the technology used to develop the software system-it is a conceptually separate entity.

Traditional approaches to software design are not sufficient when designing a large-scale distributed applications that exploit code mobility and dynamic reconfiguration of software components. In these cases, the concepts of location, distribution of components among locations, and migration of components to different locations need to be taken explicitly into account during the design stage.

The paradigms themselves are independent of particular technology, and could even be implemented of a particular technology, and could even be implemented without using mobile technology at all.

Basic concepts:

Components are the constituents of software architecture. They can be further divided into code components, that encapsulate the know-how to perform a particular computation, resource-components, that represent data or devices used during the computation, and computational components, that are active executors capable to carry out a computation, is specified by a corresponding know-how. Interactions are events that involve two or more components, e.g., a message exchanged among two or, more computational components. Sites host components and support the execution of computational components. A site represents the intuitive representation of location, Interactions among components residing at the same site are consider less

expensive than interactions taking place among components located in different sites. In addition, a computation can be actually carried out only when the know-how describing the computation, the resources used during the computation, and the computation a component responsible for execution are located at same site.

Design paradigms are described in terms of interaction patterns that define the locations and co-ordinations among the components need to perform a service. Consider a scenario where a computational component A, located at a site S_A needs the results of a service. Assuming the existence of another site S_B, which will involve the accomplishment of the service.

There are three main design paradigms exploiting code mobility:

- Remote Evaluation(RE)
- Code on Demand(COD)
- Mobile Agent(MA)

These paradigms characterized by the location of components before and after the execution of the service, by the computational component, which is responsible for execution of code, and by the location where the computation of the service actually takes place. (Table 2.1)[2]

This table shows the location of the components before and after the service execution. For each paradigm, the computational component in bold face is the one that executes the code. Components in the italics are those that have been moved.

Table 2.1 Mobile Code Paradigms

Paradigm	S_A Before	S_B Before	S_A After	S_B After
Client-Server	A	Know-how Resource B	A	Know-how Resource **B**
Remote Evaluation	Know-how A	Resource B	A	Know-how Resource **B**
Code on Demand	Resource A	Know-how B	Resource Know-how **A**	B
Mobile Agent	Know-how A	Resource	-	Know-how Resource ***A***

Client-Server:

This paradigm is well known and widely used. In this paradigm, a computational component B (the server) offering a set of services is placed at site S_B. Resources and know-how needed for service execution are hosted by site S_B as well. The client component A, located at site S_A, requests the execution of a service with an interaction with the server component B. As a response, B performs the requested service by executing the corresponding know-how and accessing the involved resources colocated with B. In general, The service produces some sort of result that will be delivered back to the client with an additional interaction.

Remote Evaluation:

In the REV paradigm, a component A has know-how necessary to perform the service but it lacks the resources required, which happen to be located at remote site S_B. Consequently, A sends the service know-how to a computational component B located at the remote site. B, in turn, executes the code using the resources available there. An additional interaction delivers the results back to A.

Code on Demand:

In the COD paradigm, component A is already able to access the resources it needs, which are co-located with it at S_A. However, no information about how to manipulate such resources is available at S_A. Thus, A interacts with a component B at S_B by requesting the service know-how, which is located at S_B as well. Second interaction takes place when B delivers the know-how to A, that can subsequently execute it.

Mobile Agent:

In the MA paradigm, the service know-how is owned by A. which is initially hosted by S_A, but some of the required resources are located on site S_B. Hence, A migrates to S_B, carrying the know-how and possibly some intermediate results. After it has moved to S_B, A completes the service using the resources available there. The mobile agent paradigm is different from the other paradigms since the associated interactions involve the mobility of an existing computational component. In other words, while in EV and COD the focus is on the transfer of code between components, In the mobile agent paradigm a whole computational component is

moved to a remote site, along with its state, the code it needs, and some resources required to perform the task

2.5 Applications of Mobile Agents:

Mobile agents can be useful for many applications.[2][3][7] Information retrieval on the network can be supported much more efficiently if an agent representing a query can move to the place where the data are actually stored, rather than having to move all of the data across the network for shifting (and subsequently discarding most of the transmission). This works especially well for non anticipated queries, i. e., the implementor of a database system cannot in general foresee everything users might want to find out and provide code to do the relevant analysis. Thus, if users have to write their own custom retrieval software, an agent-based approach can save a lot of network traffic. This is even more evident when taking into account that agents can move to other sources of information if that seems more promising. Techniques such as semantic routing—dispatching a query according to where it is most likely to be answered, rather than according to some predetermined addressing information—can be used to further boost such a system's utility and ease of use.

Another area where mobile agents can profitably be used is network management. In big networks, comprising hundreds or thousands of connected computers, operations monitoring and fault detection is very difficult and involves large amounts of logging data. It is not possible to prefabricate diagnostic programs for every eventuality, but it would be feasible to use mobile agents to keep tabs on the

system, homes in on possible trouble spots or performance bottlenecks and brings them to the attention of the maintainers.

Electronic commerce is another domain which seems amenable to mobile agents. Business on the Internet is becoming a reality, and, as standards for electronic payment are deployed, commercial 'premises' accessible via the net will probably mushroom. Mobile agents can help locate the cheapest offerings, negotiate deals or even conclude business transactions on behalf of their owners.

Finally, an important application of mobile agents concerns mobile computing. Portable computers become smaller and more powerful, but wireless access to a fixed information infrastructure is likely to stay slow and cumbersome due to restrictions on radio transmission. Besides, to minimize power consumption and transmission costs, users will not want to remain on-line while some complicated query is handled on their behalf by the fixed computing resources. Mobile agents offer a promising way out of this dilemma, users simply submit mobile agents which embody their queries and log off, waiting for the agents to deposit their results ready to be picked up at a later time.

Obviously, none of these applications absolutely require the use of mobile agents, most could be handled by stationary programs and some suitable communications paradigm such as RPC. However, this could only be done at a price of increased system (and network) load and, possibly, at the inconvenience of the users. It is also important to point out that agents are not forced to move, even though the system may allow them to do so. Some agents may be too big to move comfortably, and for others there may be no necessity. Such stationary agents could

still communicate with their mobile counterparts to take part in an agent-based system.

CHAPTER 3

AGENT CLONING

3.1 The Cloning Approach:

Cloning[4] is a possible response of an agent to overloads. Agent overloads are either to the agent's limited capacity to process current tasks or to machine overloads. Some approaches to overloads

- Task transfer: overloaded agents locate other agents, which are lightly loaded and transfer tasks to them, is very similar to load balancing.
- Agent migration: overloaded agents or agents that run on overloaded machine migrate to less overloaded machine, is closely related to process migration and to recently emerging field of mobile agents.

A main difference between load balancing and agent cloning is that while the first explicitly discusses machine loads and agent migration, the latter, in addition, considers a different type of load-the agent load.

Therefore, cloning is a superset of task transfer and agent migration: it includes them and adds to them as well. Cloning does not necessarily require migration to other machines. Rather, a new agent is created on either the local or a remote machine. Note that there may be several agents running on the same machine, and having one of them overloaded does not necessarily imply that the other are overloaded. Agent overload does not imply machine overload, and therefore local cloning (i.e., on the same machine) may be possible. Cloning takes advantage of these

idle processing capacities.

To perform cloning, an agent must reason about its own load (current and feature) and its host's load, as well as capabilities and loads of other machines and agents. Accordingly, it may decide to create a clone, pass tasks to clone, merge with other agents, or die. Merging of two agents or self-extinction of under utilized agents is an important mechanism to control agent proliferation with resulting overload of network resources.

To avoid communication overhead in trying to access and reason about remote hosts, reason regarding cloning begins by considering local cloning. When this is found infeasible or non-beneficial, the agent proceeds to reason about remote cloning. If remote cloning is decided on, an agent should be created and activated on a remote machine. Assuming that the agent has an access and a permit to work on this machine, there may be two methods to perform this cloning:

- Creating the agent locally and letting it migrate to the remote machine(similar to mobile agent)
- Creating and activating the agent on the remote machine

While the first method requires very little on the part of the remote machine, it requires mobilization properties as well as additional resource consumption. The second method, while avoiding mobilization and local resource consumption, requires that a copy of the agents code to be located on the remote machine. Similar requirements are also hold for mobile agent applications, since an agent server or agent dock required. Nonetheless, the amount of this code is small.

Since the agent 's own load and the loads of other agents may vary over time

in a non-deterministic way, the decision regarding whether and when to clone is nontrivial. In this work, a stochastic model of decision-making based on the dynamic programming used to determine the optimal timing for cloning.

Suppose a clone has been created and activated. Several questions remain with respect to this clone. These regard its autonomy, tasks, lifetime, and access to resources. Autonomy refers to independent vs. subordinate clone. Having been created and activated, an independent clone is not controlled by its creator. Therefore, such a clone will continue to exist after completion of the tasks provided by its initiator agent. Hence, a mechanism for deciding what it should do afterward is necessary. Such a mechanism must allow the clone to reason about the agent and task environment, and accordingly decide whether it should continue to work other tasks, merge with others, or perform self-extinction.

A subordinate-clone will remain under the control of its initiator. This will prevent the complications arising in the independent clone case. However, in order to mange a subordinate agent, the initiating agent must be provided with a control mechanism for remote agents. Regardless of the details of such a mechanism, it will require additional communications between the two agents, thus increasing the communications overhead of such a cloning method and the MAS's vulnerability to communication flaws. In addition, control of other agents is a partially centralized solution, which might violate rather reason for using an MAS in first place.

3.2 Cloning Initiation:

An agent should consider cloning if:

- It cannot perform all of its tasks on time by itself or decompose them so that they can be delegated to others
- There is no lightly loaded agent that can receive and perform its excess tasks (or sub tasks when tasks are decomposable)
- There are sufficient resources for creating and activating a clone agent
- The efficiency of the clone agent and the original agent is expected to be greater than that of the original agent alone

The necessary information used by an agent to decide whether and when to initiate cloning comprises parameters that describe both local and remote resources. In particular, the necessary parameters are -

- The CPU and memory loads, both internal to the agent (which result from planning, scheduling and task execution activities of the agents) and external (on the agent host and possibly on the remote hosts)
- The CPU execution speed, both local and remote
- The load on communication channels and their transfer rate, both local and remote
- The current queue of tasks, the resources required for their execution, and their deadlines
- The future expected flow of the tasks

To acquire the above information an agent must able to read the operating system variables. In addition, the agent must have self-awareness on two levels: -

- Agent-internal level: internal self-awareness should allow the agent to realize what part of the operating system retrieved values are its own properties.
- An MAS level: system-wise self-awareness should allow the agent to find, possibly via middle agents, information regarding possible resources on remote machines.

With out middle agents, servers that are located on the remote hosts can supplies such information. When such information is not available, an agent should compute the expectation values of the attributes of remote machines relying on the probability discussions either specifically by machine ID or groupwise by machine type. Pseudo-code of the cloning approach is in Appendix A.

3.3 Optimizing When to Clone:

To maximize the benefits of cloning, an agent should decide on performing it at the optimal time. Each decision regarding cloning has a value, calculated with respect to loads and distances as a function of time. Here, loads referring to processing load, memory load, and communication load. The distance between agents is the cumulative distance according to the communication route between them. An Agent A_j has a valuation function Val_j (loads, distances), where loads is a set of loads of agents and distances is a set of distances to other agents. When measuring the time in discrete units, the possible decisions of A_j can be described by decision tree. The tree includes a set of decision points, which are the nodes of the tree. The edges of the tree are decisions and each is attached a value. These values may be discounted over

time by a given discount rate r_j. The discount rate is used in cases where the agent assumes that the value of a decision is discounted over time (otherwise r_j=0). A recursive function to evaluate the decision making with dependency

$$\begin{aligned} \text{Value}(v) = \quad & 0 && \text{if no decision taken} \\ = \quad & \text{Val}(v) && \text{if decision has no dependencies} \\ = \quad & 1/(1+rj) \sum_{j=1}^{j=k} p_j \, \text{Value}(v_j) && \text{Otherwise} \end{aligned}$$

where the sum is over all the edges (v, v_j) emanating from v, and p_j is the probability of edge (v, v_j) and the corresponding decision being chosen. In this work standard dynamic programming method used to compute the optimal decision with respect to a given decision tree. For cloning mechanisms this implies a cloning timing which is optimal with respect to the available information regarding future overloads.

3.4 The Cloning Algorithm:

Among load balancing algorithms, there are two main approaches:

- Overload processors that seek other idle processors to let them perform part of the processes;
- Idle (or lightly loaded) processors that look for processes to increase their load.

These approaches are sometime combined with additional heuristics, or even merged. If these two approaches are utilized when designing a cloning algorithm for agents, considerable difficulties arise. Both approaches require that an agent locate other agents for task delegation. When using match making agents, the first approach only requires that underloaded agents advertise their capabilities; thus overloaded

agents may contact them via match making. Similarly the latter approach requires that overloaded agents may advertise their overload and required capabilities and resources. In addition, it requires that underloaded machines will be known to the overloaded agents as well, if there are no agents running on these machines. This information is not given in open system. It could be provided if each machine runs an agent whose sole task would be supplying such information. This leads to an undesirable overhead of communication and computation. In this work first approach is utilized. The cloning procedure consists of the following components:

Reasoning Before Cloning:

Includes reasoning about the (possibly dynamic) task list with respect to time restrictions and capability requirements. The consideration of the task list as well as agent capabilities, capacities, loads and machine loads results in a decision to clone or transfer tasks to already existing agents

Dividing the List of Tasks:

Includes reasoning that considers the time intervals in which overloads are expected and accordingly selects tasks to be transferred. Suppose the current and future tasks have been scheduled. At each point in time, the required resources are the sum of required resources for all of the tasks that are scheduled to be executed at this time. Figure 3.1 shows an example of the sums of three resources: cpu (p), memory (m), and communication (m), with respect to time. The maximum capacity of the agent is depicted by the threshold horizontal line (th) leveled at 6. One can observe overloads whenever any type of demand for resources crosses this threshold. A periodic overload can be observed at times 4,9,14 with period of 5 time units. Other

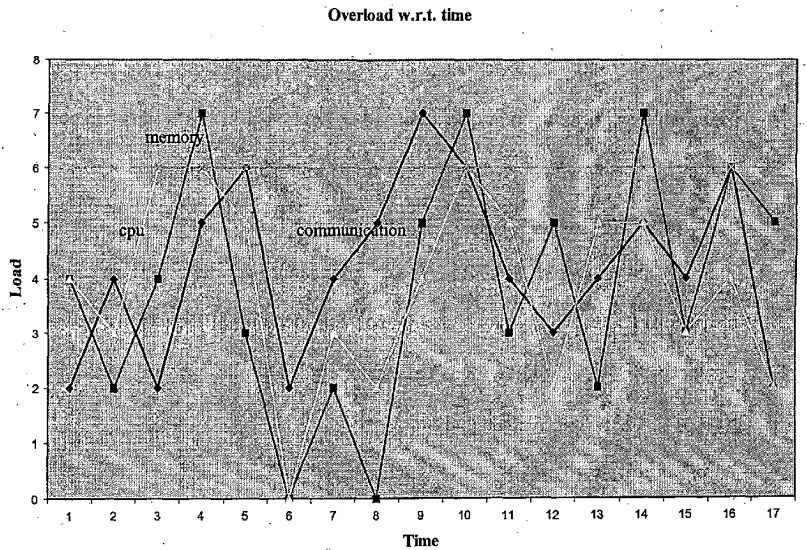

Figure 3.1 Cpu, memory, communication loads

overloads do not seem periodic. When attempting to prevent overloads, the agent first look for tasks with a period that fits the period of the overloads and puts them in the list of candidate tasks for division. After recomputing the loads, it transfers one-shot tasks if still necessary.

Cloning:

Includes the creation and activation of the clone, the transfer of tasks, and the resulting inevitable updates of connections between agents via match making. The following are the basic actions to be taken:

- Create a copy of its code. This copy, however, may have to undergo some modification.
- When cloning while performing a specific task, an agent should pass to its clone only the relevant subtasks and information necessary for the tasks passed to the clone.

Otherwise, the clone may face the same overload problem as its creator. This is because the clone will have the same set of tasks as its creator had. Note that in contrast to the typical approach to agent migration, the cloning paradigm does not require the transfer of an agent state. The only transfer necessary is of the set of tasks to be performed by the clone.

Reasoning After Cloning:

Collects information regarding the benefits of the cloning and environmental properties (such as task stream distribution) and statically analyzes them as a means of learning for future cloning.

While the reasoning of whether to initiate cloning is performed continually

(i.e., when there are changes in the task schedule or if previous attempts to clone have failed), the task cloning itself is a one-shot procedure.

3.4 Merging of Agents:

Suppose a clone has been created and activated. Having been created and activated, an independent clone is not controlled by its creator, because of agent's autonomy. Therefore such a clone will continue to exist after completion of the tasks provided by its initiator agent. Hence a mechanism for what it should do afterward is necessary. Such a mechanism must allow the clone to reason about the agent and task environment, and accordingly decide whether it should do or continue to work on other tasks, merge with others, or perform self-extinction.

An agent considers merging only if it is underloaded continuously. Before considering merging, agent has to reason about its tasks over a period of time, because agent receives the tasks randomly (gaussian). So agent analyzes the task environment over a period of time. To analyze, agent stores the information about the loads and tasks. Then using this information agent expects the task list. In this work, back propagation network is used to expect the task list. The neural network is trained by using loads and tasks information. After this, the agent reasons about the underload using the expected task list.

In case the agent is underloaded over a period of time, then agent analyzes the agent environment, because one agent is underloaded does not mean total agents are underloaded. So it is necessary to consider the host's overload as well as the system's overload. In the case, if the system and host environment is underloaded then agent may not consider merging. And in the second case, the system is underloaded and the

host's is overoaded then agent migrates to remote host. In the last case, the host's is underloaded or overloaded, and the system is overloaded the agent does not consider merging, but the agent waits for the tasks from the overloaded agents. These peculiar situations may not be possible, but it is necessary to define every situation.

Suppose, if the agent has decided to merge with other agents then agent has to find out the other agents, which are also underloaded, and wants to merge. If there is no such agent then agent has to reason about whether agent should die or not, otherwise the mergable, underloaded agents should find out leader, so as to transfer the responsibilities. Generally leader agent is the one who has maximum responsibilities. The agent transfers the tasks and responsibilities to leader and releases the resources.

The agent acquires the resources from the host to perform the services. An agent provides the services by performing the tasks which either it generates by itself, or users or other agents delegate to it. If the agent is underloaded continuously the resources acquired by the agent is not used. So the utilization of the network resources is low. So to control the performance degradation of the system, merging of agents is needed. Pseudo-code of this approach is in Appendix B.

3.5 Merging approach:

To perform merging, In this work following approach is used. In case the agent is not overloaded, then agent reasons about the underloads. In the event the agent is underloaded, then the underload count incremented. In the case, the underload count exceeds the thresh hold count then agent tries to find out the list of agents who also desire to merging, after setting the merge flag. If the mergable agent

list is empty then agent considers about the responsibilities. Suppose, if the agent there are no responsibilities then agent dies after releasing the resources, otherwise nothing can be done. But there is another approach also possible. In this approach, the agent reasons about the system overload with respect to the number of agents and number of tasks. The agent finds out the number of tasks the system receives and the number of agents in the system. In this work it is assumed that the system able to provide the above information . The number of agents in the system above thresh hold and the number of tasks the system is receiving is below average, then agent will do the following mechanism. The agent will find out the list of agents, which are underloaded, but not mergable. The agent transfers the tasks to that agent (responsibilities) and dies after releasing the resources.

In the event the agent(mergable) list is not empty then those agents elects one leaders and all the agents transfer the responsibilities to that agent. The agent selection depends on the communication costs from other agents to that selected agent and the responsibility of that selected agent. In the case, if there is no underload then agent unsets the merge flag and sets the value of to 0.

To evaluate the underload, the agent uses the back propagation neural network. Back propagation algorithm is given in Appendix C. Suppose, the required resources for the expected tasks is below the threshold resources then agent considered to be under loaded.

But the basic disadvantage of the above method is that agent only depends on the neural network and the thresh hold. But it is necessary to develop a theoretical framework and consider the agent approach.

CHAPTER 4

DESIGN AND IMPLEMENTATION

To examine the properties of the cloning mechanism and its advantages, a simulation was performed. The simulation shows that cloning increases (on average) the performance of the MAS. That is, the performance enhancement as a result of cloning outweighs the efforts put into cloning.

4.1 Method of Simulation:

Each agent represented by an agent thread that simulated the resource consumption and tasks queue of a real agent. The simulated agent has a reasoning-for-cloning method, which, according to the resource consumption parameters and task queue, reasons about cloning. As a result of this reasoning it may create a clone by activating another agent thread (either locally or remotely). Information collected during the simulation is the usage of CPU and the memory and communication consumption of the agents. Each agent thread receives a stream of tasks according to a given distribution. For each task it creates a task object that consumes time and memory and requires communication. Some of these task objects are passed to the clone agent thread. The simulation was performed with and without cloning to allow comparison.

An agent thread in simulation must be subject to CPU, memory, and communication consumption similar to those consumed by an agent it models in the MAS. The program written in Java[11][12] under Linux operating system.

4.2 Simulation Parameters:

The simulated agent system has the following parameters:

- Number of agents: 10-20
- Number of Clones allowed: 10
- Number of tasks dynamically arriving at the system: up to 1000
- Task distribution with respect to the required capabilities and resources for execution: normal distribution, where 10 percent of the tasks are beyond the ability of the agents to perform their particular deadlines.
- Agent capacity: an agent can perform 20 average tasks (i.e. requiring the average amount of resources) simultaneously.

4.3 PROGRAM CLASSES:

Classes:

Agent Manager:-

This class creates agents and starts the agents. It also assigns the tasks to the agents.

Attributes: -

agentThread:- it provides references to all agents

totalNumOfAgents:- it counts the num of agents

totalTasks:- tasks assigned to the Multi Agent System

out:- reference to log file

net:- a reference to network

Operations:-

createAgents:- this method creates agents and starts the threads

assignTasks:- this method assigns the tasks to the system for service

increaseAgents:- agents registers the newly created cloned agent.

run:- main controling method of agent manager

Agents:

This is the main class of simulation. It simulates agent.

Attributes:

Some important attributes are-

cpu , memory, communication:- resource parameters of agent

cpuLoad, memoryLoad, communicationLoad:- the loads of the agent

agentID:- the id of agent for communication in the system

hostID:– the host id on which agent has hosted

cloneFlag:– flag whether cloning is allowed or not

overload:– flag denotes whether agent is overloaded or not

out:– log file reference

random:– gaussian random variable

taskList:– list of tasks for servicing of agent, assigned by agent manager

taskSplit:- used for splitting the tasks so as to assign to remote agent

Operations:-

run:- main method of agent thread

reasonForCloning:- reasons about loads of itself and system.

reasonAboutOverloads: - finds the present and future loads of agent, and

reasons about overload

findUnderloadedAgents: - find the under loaded agents for task transfer

reasonForTaskSplit: - It will find out the periods of tasks and it splits the tasks according to overload, otherwise it will split tasks serially until load is decreased

transferTasks: - it transfers the from the taskSplit list to the remote agent taskList for servicing

canCloneLocally: - it will find out whether local cloning is possible or not

cloneLocally: - it create the agent locally

canCloneRemotely: - finds whether any remote cloning is possible or not

cloneRemotely: - it creates the agent remotely (remote cloning)

reasonAfterCloning: - reasons for further cloning and the feasibility of cloning

getExpectedTaskFlow: – it finds the expected tasks using back propagation network

getSelfCapacity: - finds the self capacity of agent

totalResourcesRequired: - it find out the resources required for tasks(present and future)

findPeriodsOfTasks: - it find out the periods of tasks

Task: -

It represents a task object, which consumes resources.

Attributes: -

cpu, memory, and communication: - represents the resources required r a task object

runningFlag: - denotes the whether a task is running or not

Operations: -

start: - it starts the task object for servicing

stop: - it stops the servicing of task object

isRunning: - finds whether an object is running or not

isCompleted: - find whether service of task is completed or not

Host: -

It represents a host in Multi Agent System:

Attributes: -

cpu, memory, and communication: - are the resources available at host

agentList: - it is a hash table for storing the agents list who are under service

hostID: - it is id of host

Opeartions: -

recoverResources: - releases the resources acquired by an agent

requsetForSevices: - request the host for resources, called by agent

requestForCloning: - request host for cloning (remote cloning)

CHAPTER 5

CONCLUSION

5.1 Discussion of Results:

The results of the simulation are depicted in Figure 5.1. The graph show that for small number of tasks (0-100) a system which practices cloning performs (almost) as well as a system with no cloning (although difficult to see in the graph, the performance is slightly lower due to reasoning costs). However, when the number of tasks increases the cloning system performs much better. Nonetheless, beyond some threshold (around 350 tasks) even the cloning cannot help. Note that in the range 150-350 tasks cloning results in task performance close to optimal, where optimality refers to the case in which all of the available resources are efficiently used for task performance.

5.2 Concluding Remarks:

Agent cloning is the action of creating and activating a clone agent (locally or remotely) to perform some or all of an agent's tasks. Cloning is performed when an agent perceives or predicts an overload, thus increasing the ability of an MAS to perform tasks. In this work agent cloning is used as a means for balancing the loads and improving the task performance of an MAS running on several machines. Methods of implementation are also studied and tested these methods by simulation. It is found that for large number of tasks, cloning significantly increases the

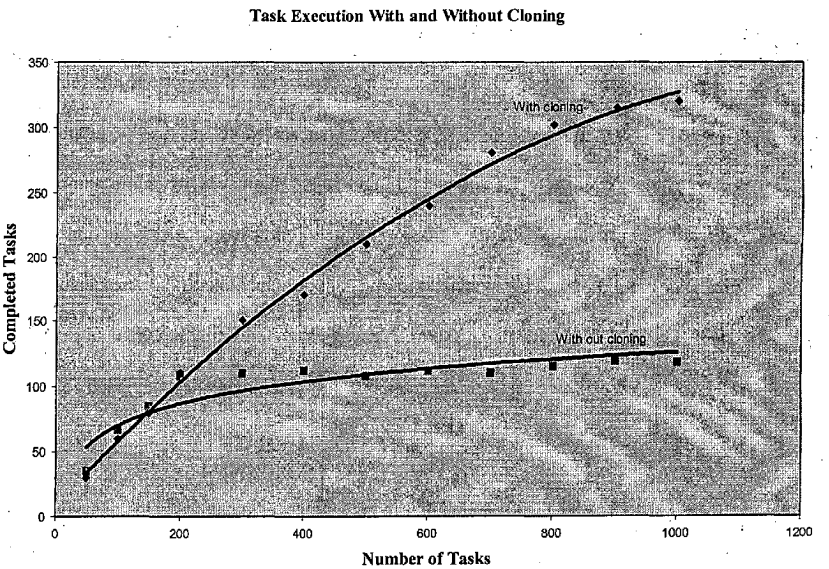

Figure 5.1 Task Execution with and without cloning

portion of tasks performed by an MAS. In an MAS where tasks require information gathering on the web, the additional reasoning needed for cloning is small compared to task execution requirements. Merging of two agents or self-extinction of under utilized agents proliferation with resulting overload of network resources is also discussed.

5.3 Scope of Future work:

In this work agent cloning used as a means for improving the performance of an MAS. To perform cloning, an agent must reason regarding its own load, and its host's load, as well as capabilities and loads of other machines and agents.

A cloned agent will continue to exist after completion of tasks provided by its initiator agent. Hence, a mechanism for what it should do afterward is necessary. In this regard, following mechanism implemented in this work. The cloned agent, after performing all the tasks, waits for the tasks. If the agent not receives the tasks from other agent or user, for threshold duration of the time, it will merge with other agent or dies. But it is necessary to develop a theoretical analysis to find out the optimal time for merging or self-extinction.

Agent cloning subsumes the task transfer and agent migration, so this mechanism can be embedded in the existing systems so as to study practical difficulties of the cloning.

One more important thing, in this work only three parameters cpu, memory, communication are considered in simulation. This can be extended to other parameters.

REFERENCES

[1] "Software Agents: Overview", by Shaw Green, Leon Hurst and others available from www.cs.tcd.ie/researc_group/aig

[2] Fuggetta , G.Pieetro, G. Vigna "Understanding Code Mobility "IEEE Transactions on Software Engg., May 1998, pp.342-361

[3] V.A. Pham, and A. Karmouvh, "Mobile Software Agents Overview" IEEE commun. Magazine, July 1998,pp.26-37

[4] Onn Shehory, Katia Sycara, P.Chalasani, and Soesh Jha, "Agent Cloning: An Approach to Agent Mobility and Resource Allocation" IEEE Communications, July, 1998, pp58-67.

[5] G.Agha, "Actors and Agents", IEEE Concurrency, April-June, 1998, pp.24-28

[6] K.Sycara , A.Pannu, M.Williamson, and D. Zeng, "Distributed intelligent agents," IEEE Expert, Dec., 1996, pp.36-45

[7] Anselm Lingnau Oswald Drobnik, "An infrastructure for Mobile Agents: Requirements and architecture" Fachbereich Informatik (Telematik), Johann Wolfgang Goethe-University " at Frankfurt am Main, Germany

[8] N.M.Karnik, A.R. Tripathi., "Design Issues In mobile Agent Programming Systems,", IEEE Concurrency, July-September 1998, pp.52-61

[9] D.Chess, B. Grosof, and C. Harrison, "Itinerant agent for mobile computing", IEEE Pers. Communications Magazine, June,1995

[10] Adam Blum, "Neural Network in C++", First Edition, John Wiley & Sons, Inc.

[11] K.Decker , K.Sycara, and M Williamson, "Middle Agents for the Internet,"

Proc. IJCAI '97, Nagoya, Japan, 1997

[11] K.Decker, and K.Sycara ., "Designing behaviours for information agents," W. Lewis Johnson, Ed., Proc. 1st Int'l Conf. Autonomous Agents, ACM Press 1997

[12] Sun Micro Systems, "The Java Language: An Overview", Technical Report, Sun Micro Systems,1994

[13] Gosling, J. and McGilton, H. (1995). "The Java Language Environment: A White Paper", Technical Report, Sun Microsystems.

APPENDIX A

Cloning Pseudo-Code:

```
//Main reasoning and cloning protocol
for each time interval t{
        overloadAt[t]=reasonAboutOverloads(t);//current and future;
        If(overloadAt[t]) {
                CanPassTasks[t]=findUnderloadedAgents(t);
                If(canPassTasks[t]){
                        ReasonForTaskSplit();
                        TransferTasks();
                }else if(canCloneLocally()){
                        cloneLocally();
                        reasonForTaskSplit();
                        transferTasks();
                }else if(canCloneRemotely()){
                        cloneRemotely();
                        reasonForTaskSplit();
                        transferTasks();
                }else sorry("can't split or clone");

//The reasoning methods
reasonAboutOverloads(t)
{
```

```
        getCurrentTaskList();
        getExpectedTaskFlow(t);
        requiredAt[t]=calculateRequiredResourcesAt(t);// current and future
        selfCapacity=getSelfCapacity(t);
        if(selfCapacity >=requiredAt[t])
                return true;
        else
                return false;
}
```

247866.

APPENDIX B

Merging Pseudo-Code:

```
for each time interval t {
        underloadAt [t]=reasonAboutUnderloads(t);
        if(underloadAt[t]){
                increment(underloadCount);
                if(underloadCount > threshholdCount) {
                        mergeFlag=true;
                        canMerge=findMergableAgentsList();
                        if(canMerge){
                                leader=selectLeader();
                                transferResponsibilities();
                                releaseResources();
                                stop();// dying of an agent
                        }else if(no responsibilities){
                                releaseResources();
                                stop(); //dying of an agent
                        }
        }else {
        underloadCount=0;
        mergeFlag=false;
        }
}
```

```
// reasoning about under loads
reasonAboutUnderloads()
{
        if(overloadAt[t])
                return false;
        getCurrentTaskList();
        if(!empty(taskList))
                return false;
        getExpectedTaskFlowUsingNeural(t);
        requiredAt[t] = calculaeRequiredResources(t);
        selfCapacity=getSelfCapacity();
        if(requiredAt[t] < selfCapacity/2.5)
                return false;
        else
                return true;
}
```

APPENDIX C

Back propagation Algorithm:

The back propagation model is applicable to a wide class of problems. It is certainly the predominant supervised training algorithm. The algorithm is

2.3.1 Encoding:

Assign random values between –1 and +1 to the weights between the input and hidden layers, the weights between the hidden and output layers, and the thresholds for the hidden-layer and output-layer neurons.

Forward Pass:

1 Compute the hidden-layer neuron activation:

h=F(iW1)

where **h** is the vector of hidden-layer neurons, **i** the vector of input –layer neurons, and **W1** the weight matrix between the input and hidden layers.

2 Compute the output-layer neuron activations:

o=F(hW2)

where **o** represents the output layer, **h** the hidden layer, **W2** the matrix of synapses connecting the hidden and output layers, and **F()** is sigmoid activation function.

Backward Pass:

3 Compute the output-layer error (the difference between the target and the observed output):

d = o(1-o)(o-t)

where **d** is the vector of errors for each output neuron, **o** is the output-layer vector, and **t** is the target (correct) activation of output-layer.

4 Compute the hidden-layer error:

$$\mathbf{e = h(1-h)W2d}$$

where **e** is the vector of errors for each hidden-layer neuron.

5 Adjust the weights for the second layer of synapses:

$$\mathbf{W2 = W2 + \Delta W2}$$

where delta **W2** is a matrix representing the change in matrix **W2**. It is computed as follows:

$$\Delta \mathbf{W2}_t = \alpha\, \mathbf{hd} + \theta\, \Delta \mathbf{W2}_{t-1}$$

where α is the learning rate and θ is the momentum factor

6 Adjust the weights for the first layer of synapses:

$$\mathbf{W1 = W1 + W1}_t$$

where

$$\mathbf{W1}t = \alpha\, \mathbf{ie} + \theta\, \Delta \mathbf{W1}_{t-1}$$

Repeat steps **1** to **6** on all patterns until the output-layer error (vector **d**) is within the specified tolerance for each pattern and for each neuron

APPENDIX D

SOFTWARE LISTING

```
/*
        This simple extension of the java.awt.Frame class
        contains all the elements necessary to act as the
        main window of an application.
*/
import java.io.*;
import java.awt.*;

public class cloneFrame extends Frame implements Constants
{
  static cloneFrame clone;
  public cloneFrame()
  {

    setLayout(new BorderLayout(0,0));
    setVisible(false);
    setSize(405,364);
    controlPanel = new java.awt.Panel();
    controlPanel.setLayout(new FlowLayout(FlowLayout.CENTER,5,5));
    controlPanel.setBounds(0,0,434,33);
    controlPanel.setBackground(new Color(-12948347));
    add("North", controlPanel);
    startButton = new java.awt.Button();
    startButton.setLabel("Start");
    startButton.setBounds(79,5,39,23);
    startButton.setBackground(new Color(12632256));
    controlPanel.add(startButton);
    stopButton = new java.awt.Button();
    stopButton.setLabel("Stop");
    stopButton.setBounds(123,5,39,23);
    stopButton.setBackground(new Color(12632256));
    controlPanel.add(stopButton);
    stopButton.setEnabled(false);
    aboutButton = new java.awt.Button();
    aboutButton.setLabel("About...");
    aboutButton.setBounds(167,5,54,23);
    aboutButton.setBackground(new Color(12632256));
    controlPanel.add(aboutButton);
    quitButton = new java.awt.Button();
    quitButton.setLabel("Quit");
    quitButton.setBounds(226,5,36,23);
    quitButton.setBackground(new Color(12632256));
    controlPanel.add(quitButton);
    Group1 = new CheckboxGroup();
    cloneRadioButton = new java.awt.Checkbox("Cloneable", Group1,
                false);
```

```
cloneRadioButton.setBounds(267,5,87,23);
controlPanel.add(cloneRadioButton);
cloneRadioButton.setEnabled(false);
editPanel = new java.awt.Panel();
GridBagLayout gridBagLayout;
gridBagLayout = new GridBagLayout();
editPanel.setLayout(gridBagLayout);
editPanel.setBounds(0,33,434,306);
editPanel.setBackground(new Color(-1254719));
add("Center", editPanel);
hostLabel = new java.awt.Label("Number Of Hosts");
hostLabel.setBounds(60,24,96,27);
hostLabel.setForeground(new Color(0));
GridBagConstraints gbc;
gbc = new GridBagConstraints();
gbc.gridx = 0;
gbc.gridy = 0;
gbc.gridheight = 2;
gbc.fill = GridBagConstraints.NONE;
gbc.insets = new Insets(24,60,0,0);
gbc.ipadx = -14;
gbc.ipady = 4;
((GridBagLayout)editPanel.getLayout()).setConstraints(hostLabel,
                    gbc);
editPanel.add(hostLabel);
agentLabel = new java.awt.Label("Number Of Agents");
agentLabel.setBounds(60,60,108,26);
gbc = new GridBagConstraints();
gbc.gridx = 0;
gbc.gridy = 2;
gbc.gridwidth = 2;
gbc.fill = GridBagConstraints.NONE;
gbc.insets = new Insets(9,60,0,0);
gbc.ipadx = -7;
gbc.ipady = 3;
((GridBagLayout)editPanel.getLayout()).setConstraints(agentLabel,
            gbc);
editPanel.add(agentLabel);
hostField = new java.awt.TextField();
hostField.setBounds(186,12,148,36);
hostField.setBackground(new Color(16777215));
gbc = new GridBagConstraints();
gbc.gridx = 2;
gbc.gridy = 0;
gbc.gridwidth = 3;
gbc.fill = GridBagConstraints.NONE;
```

```
gbc.insets = new Insets(12,12,0,0);
gbc.ipadx = 124;
gbc.ipady = 13;
((GridBagLayout)editPanel.getLayout()).setConstraints(hostField,
                    gbc);
editPanel.add(hostField);
agentField = new java.awt.TextField();
agentField.setBounds(185,60,150,28);
agentField.setBackground(new Color(16777215));
gbc = new GridBagConstraints();
gbc.gridx = 2;
gbc.gridy = 2;
gbc.gridwidth = 4;
gbc.gridheight = 2;
gbc.fill = GridBagConstraints.NONE;
gbc.insets = new Insets(9,12,0,0);
gbc.ipadx = 126;
gbc.ipady = 5;
((GridBagLayout)editPanel.getLayout()).setConstraints(agentField,
                    gbc);
editPanel.add(agentField);
createNetButton = new java.awt.Button();
createNetButton.setLabel("Create Net");
createNetButton.setBounds(96,108,85,36);
createNetButton.setBackground(new Color(12632256));
gbc = new GridBagConstraints();
gbc.gridx = 0;
gbc.gridy = 4;
gbc.gridwidth = 3;
gbc.fill = GridBagConstraints.NONE;
gbc.insets = new Insets(20,96,0,0);
gbc.ipadx = 12;
gbc.ipady = 13;
((GridBagLayout)editPanel.getLayout()).setConstraints(
                        createNetButton, gbc);
editPanel.add(createNetButton);
createNetButton.setEnabled(false);
createAgentsButton = new java.awt.Button();
createAgentsButton.setLabel("Create Agents");
createAgentsButton.setBounds(228,108,85,40);
createAgentsButton.setBackground(new Color(12632256));
gbc = new GridBagConstraints();
gbc.gridx = 3;
gbc.gridy = 4;
gbc.gridheight = 2;
gbc.fill = GridBagConstraints.NONE;
```

```
gbc.insets = new Insets(20,47,0,0);
gbc.ipadx = -7;
gbc.ipady = 17;
((GridBagLayout)editPanel.getLayout()).setConstraints(createAgen
        tsButton, gbc);
editPanel.add(createAgentsButton);
createAgentsButton.setEnabled(false);
resultArea = new java.awt.TextArea();
resultArea.setEditable(false);
resultArea.setBounds(0,195,434,111);
resultArea.setForeground(new Color(-8563541));
resultArea.setBackground(new Color(8421504));
gbc = new GridBagConstraints();
gbc.gridx = 0;
gbc.gridy = 6;
gbc.gridwidth = 7;
gbc.weightx = 1.0;
gbc.weighty = 1.0;
gbc.fill = GridBagConstraints.BOTH;
gbc.insets = new Insets(47,0,0,0);
gbc.ipadx = -8;
gbc.ipady = -76;
((GridBagLayout)editPanel.getLayout()).setConstraints(resultArea
, gbc);
editPanel.add(resultArea);
setTitle("Agent Cloning");
setResizable(false);

//REGISTER_LISTENERS
SymWindow aSymWindow = new SymWindow();
this.addWindowListener(aSymWindow);
SymAction lSymAction = new SymAction();
startButton.addActionListener(lSymAction);
createNetButton.addActionListener(lSymAction);
stopButton.addActionListener(lSymAction);
createAgentsButton.addActionListener(lSymAction);
SymText lSymText = new SymText();
hostField.addTextListener(lSymText);
agentField.addTextListener(lSymText);
quitButton.addActionListener(lSymAction);
SymItem lSymItem = new SymItem();
cloneRadioButton.addItemListener(lSymItem);

openLogFile();
}
Network net=null;
```

```
    PrintWriter out=null;
    void openLogFile()
    {
      if(out!=null)
      out.close();
  try{
        out=new PrintWriter(new FileOutputStream("output.txt"));
      }catch(Exception e){
          setVisible(false);
          dispose();
          System.out.println(e);
          System.exit(1);
        }
  if(out==null)
      System.out.println("Unable to open the file ");
    }
    public cloneFrame(String title)
    {
          this();
          setTitle(title);
    }

/**
 * Shows or hides the component depending on the boolean flag b.
 * @param b  if true, show the component; otherwise, hide the component.
 * @see java.awt.Component#isVisible
 */
public void setVisible(boolean b)
    {
          if(b)
          {
                setLocation(50, 50);
          }
          super.setVisible(b);
    }

    static public void main(String args[])
    {
      clone=new cloneFrame();
//    clone.openLogFile();
      clone.setVisible(true);
    }

    public void addNotify()
    {
    // Record the size of the window prior to calling parents addNotify.
```

```
        Dimension d = getSize();
        super.addNotify();
        if (fComponentsAdjusted)
                return;

        // Adjust components according to the insets
        setSize(insets().left + insets().right + d.width, insets().top +
insets().bottom + d.height);
        Component components[] = getComponents();
        for (int i = 0; i < components.length; i++)
        {
                Point p = components[i].getLocation();
                p.translate(insets().left, insets().top);
                components[i].setLocation(p);
        }
        fComponentsAdjusted = true;
}

// Used for addNotify check.
boolean fComponentsAdjusted = false;

//DECLARE_CONTROLS
java.awt.Panel controlPanel;
java.awt.Button startButton;
java.awt.Button stopButton;
java.awt.Button aboutButton;
java.awt.Button quitButton;
java.awt.Checkbox cloneRadioButton;
CheckboxGroup Group1;
java.awt.Panel editPanel;
java.awt.Label hostLabel;
java.awt.Label agentLabel;
java.awt.TextField hostField;
java.awt.TextField agentField;
java.awt.Button createNetButton;
java.awt.Button createAgentsButton;
java.awt.TextArea resultArea;
//

//DECLARE_MENUS
//

class SymWindow extends java.awt.event.WindowAdapter
{
        public void windowClosing(java.awt.event.WindowEvent event)
        {
```

```
                Object object = event.getSource();
                if (object == cloneFrame.this)
                        Frame1_WindowClosing(event);
        }
}

void Frame1_WindowClosing(java.awt.event.WindowEvent event)
{
        setVisible(false);      // hide the Frame
        dispose();                      // free the system resources
        System.exit(0);                 // close the application
}

class SymAction implements java.awt.event.ActionListener
{
        public void actionPerformed(java.awt.event.ActionEvent event)
        {
                Object object = event.getSource();
if (object == startButton)
                        startButton_ActionPerformed(event);
                else if (object == createNetButton)
                        createNetButton_ActionPerformed(event);
                else if (object == stopButton)
                        stopButton_ActionPerformed(event);
                else if (object == createAgentsButton)
                        createAgentsButton_ActionPerformed(event);
                else if (object == quitButton)
                        quitButton_ActionPerformed(event);
        }
}


void startButton_ActionPerformed(java.awt.event.ActionEvent event)
{
        // to do: code goes here.
        startButton.enable(false);
        hostField.enable(true);
        agentField.enable(true);
        createNetButton.enable(true);
        cloneRadioButton.enable(true);
        stopButton.enable(true);
        //{{CONNECTION
        // Set the Button's action command... Get Frame title
        startButton.setActionCommand(getTitle());
        //}}
}
```

```
void aboutButton_MouseClicked(java.awt.event.MouseEvent event)
{

        (new AboutDialog(this, true)).setVisible(true);
        //CONNECTION
        // Set the Button's action command... Get Frame title
        aboutButton.setActionCommand(getTitle());
}


void createNetButton_ActionPerformed(java.awt.event.ActionEvent event)
{

        if(numOfHosts>15 || numOfHosts<5)
                numOfHosts=NUMHOSTS;
        hostField.setText(String.valueOf(numOfHosts));
        createNetButton.enable(false);
        createAgentsButton.enable(true);
        net=new Network(out,numOfHosts);
        net.createNetwork();
        hostField.enable(false);

        GraphPanel g;
        g=new GraphPanel(net,out);
        g.resize(400,400);
        g.show();
        // Set the Button's action command... Get Frame title
        createNetButton.setActionCommand(getTitle());
        //}}

}

DrawPanel drawPanel=null;
void stopButton_ActionPerformed(java.awt.event.ActionEvent event)
{

net=null;
if(agentManager!=null)
        agentManager.stopAgents();
if(drawPanel!=null)
        drawPanel.stop();
drawPanel=null;
agentManager=null;
startButton.enable(true);
```

```
hostField.enable(false);
agentField.enable(false);
createNetButton.enable(false);
createAgentsButton.enable(false);
cloneRadioButton.enable(false);
stopButton.enable(false);
System.gc();

        //{{CONNECTION
        // Set the Button's action command... Get Frame title
        stopButton.setActionCommand(getTitle());
        //}}
}
AgentManager agentManager=null;
void createAgentsButton_ActionPerformed(java.awt.event.ActionEvent
                event)
{
        if(numOfAgents<5 || numOfAgents>15)
                numOfAgents=NUMOFAGENTS;
        agentField.setText(String.valueOf(numOfAgents));
        createAgentsButton.enable(false);
        clonableFlag=cloneRadioButton.getState();
        cloneRadioButton.enable(false);
        Agents.clonableFlag=clonableFlag;
    agentField.enable(false);
agentManager=new AgentManager(net,out,numOfAgents,true);
    agentManager.start();
drawPanel=new DrawPanel();
    drawPanel.setVisible(true);
drawPanel.start();
        createAgentsButton.setActionCommand(getTitle());
}

class SymText implements java.awt.event.TextListener
{
        public void textValueChanged(java.awt.event.TextEvent event)
        {
                Object object = event.getSource();
                if (object == hostField)
                        hostField_TextValueChanged(event);
                else if (object == agentField)
                        agentField_TextValueChanged(event);
        }
}
int numOfHosts=NUMHOSTS;
int numOfAgents=NUMOFAGENTS;
```

```
void hostField_TextValueChanged(java.awt.event.TextEvent event)
{
String str=hostField.getText();
        try{
        numOfHosts=Integer.parseInt(str);
        out.println("Num of Hosts"+numOfAgents);
        catch(NumberFormatException e){
        System.out.println("exception in formattin"+e);
            numOfHosts=NUMHOSTS;
        hostField.setText(String.valueOf(numOfHosts));
        }catch(NullPointerException e){
        }

        // Show the TextField
        agentField.setVisible(true);
        //}}
}

void agentField_TextValueChanged(java.awt.event.TextEvent event)
{
        String str=agentField.getText();
        try{
        numOfAgents=Integer.parseInt(str);
        out.println("Num of Hosts"+numOfAgents);
        }catch(NumberFormatException e){
        System.out.println("exception in formattin"+e);
        numOfAgents=NUMOFAGENTS;
        hostField.setText(String.valueOf(numOfAgents));
        }catch(NullPointerException e){
        }

        // Show the TextField
        agentField.setVisible(true);
        //}}
}

void quitButton_ActionPerformed(java.awt.event.ActionEvent event)
{
        (new QuitDialog(this,true)).setVisible(true);
        // Set the Button's action command... Get Frame title
        quitButton.setActionCommand(getTitle());
        //}}
}

boolean clonableFlag=false;
class SymItem implements java.awt.event.ItemListener
```

```
        {
                public void itemStateChanged(java.awt.event.ItemEvent event)
                {
                        Object object = event.getSource();
                        if (object == cloneRadioButton)
                                cloneRadioButton_ItemStateChanged(event);
                }
        }

        void cloneRadioButton_ItemStateChanged(java.awt.event.ItemEvent event)
        {
                // Show the Checkbox
                cloneRadioButton.setVisible(true);
                //}}
        }
}

s/*
        A basic extension of the java.awt.Dialog class
 */

import java.awt.*;

public class AboutDialog extends Dialog {
        public AboutDialog(Frame parent, boolean modal)
        {
                super(parent, modal);

                //INIT_CONTROLS
                setLayout(null);
                setVisible(false);
                setSize(300,250);
                titleLabel = new java.awt.Label("Agent Cloning: A Mechanism to
                                load transfer");
                titleLabel.setBounds(40,35,248,21);
                add(titleLabel);
                okButton = new java.awt.Button();
                okButton.setLabel("OK");
                okButton.setBounds(168,108,66,27);
                add(okButton);
                setTitle("About Thesis");
                setResizable(false);
                //REGISTER_LISTENERS
                SymWindow aSymWindow = new SymWindow();
                this.addWindowListener(aSymWindow);
                SymAction lSymAction = new SymAction();
```

```
        okButton.addActionListener(lSymAction);
}
// constrcutor
public AboutDialog(Frame parent, String title, boolean modal)
{
        this(parent, modal);
        setTitle(title);
}

public void addNotify()
{
// Record the size of the window prior to calling parents addNotify.
    Dimension d = getSize();
        super.addNotify();
        // Only do this once.
        if (fComponentsAdjusted)
                return;

        // Adjust components according to the insets
        setSize(insets().left + insets().right + d.width, insets().top +
                insets().bottom + d.height);
        Component components[] = getComponents();
        for (int i = 0; i < components.length; i++)
        {
                Point p = components[i].getLocation();
                p.translate(insets().left, insets().top);
                components[i].setLocation(p);
        }
        // Used for addNotify check.
        fComponentsAdjusted = true;
}

public void setVisible(boolean b)
{
        if (b)
        {
                Rectangle bounds = getParent().bounds();
                Rectangle abounds = bounds();
                move(bounds.x + (bounds.width - abounds.width)/ 2,
                 bounds.y + (bounds.height - abounds.height)/2);
        }

        super.setVisible(b);
}

//{{DECLARE_CONTROLS
```

```
        java.awt.Label titleLabel;
        java.awt.Button okButton;

    // Used for addNotify check.
        boolean fComponentsAdjusted = false;

        class SymWindow extends java.awt.event.WindowAdapter
        {
                public void windowClosing(java.awt.event.WindowEvent event)
                {
                        Object object = event.getSource();
                        if (object == AboutDialog.this)
                                AboutDialog_WindowClosing(event);
                }
        }

        void AboutDialog_WindowClosing(java.awt.event.WindowEvent event)
        {
          dispose();
        }

        class SymAction implements java.awt.event.ActionListener
        {
                public void actionPerformed(java.awt.event.ActionEvent event)
                {
                        Object object = event.getSource();
                        if (object == okButton)
                                okButton_Clicked(event);
                }
        }

        void okButton_Clicked(java.awt.event.ActionEvent event)
        {
                // Clicked from okButton Hide the Dialog
          dispose();
        }
}


/*
        A basic extension of the java.awt.Dialog class
 */

import java.awt.*;
import java.awt.event.*;
```

```
public class QuitDialog extends Dialog
{
        public QuitDialog(Frame parent, boolean modal)
        {
                super(parent, modal);

                setLayout(null);
                setSize(insets().left + insets().right + 337,insets().top +
                        insets().bottom + 135);
                yesButton = new java.awt.Button(" Yes ");
                yesButton.setBounds(insets().left + 72,insets().top + 80,79,22);
                yesButton.setFont(new Font("Dialog", Font.BOLD, 12));
                add(yesButton);
                noButton = new java.awt.Button("  No  ");
                noButton.setBounds(insets().left + 185,insets().top + 80,79,22);
                noButton.setFont(new Font("Dialog", Font.BOLD, 12));
                add(noButton);
                label1 = new java.awt.Label("Do you really want to \
                                        quit?",Label.CENTER);
                label1.setBounds(78,33,180,23);
                add(label1);
                setTitle("A Basic Application - Quit");
                setResizable(false);
                //{{REGISTER_LISTENERS
                SymWindow aSymWindow = new SymWindow();
                this.addWindowListener(aSymWindow);
                SymAction lSymAction = new SymAction();
                noButton.addActionListener(lSymAction);
                yesButton.addActionListener(lSymAction);
                //}}
        }

        public void addNotify()
        {
    // Record the size of the window prior to calling parents addNotify.
        Dimension d = getSize();
                super.addNotify();
                if (fComponentsAdjusted)
                        return;

                // Adjust components according to the insets
                setSize(insets().left + insets().right + d.width, insets().top +
                                insets().bottom + d.height);
                Component components[] = getComponents();
                for (int i = 0; i < components.length; i++)
                {
```

```
                Point p = components[i].getLocation();
                p.translate(insets().left, insets().top);
                components[i].setLocation(p);
            }
            fComponentsAdjusted = true;
        }

        public QuitDialog(Frame parent, String title, boolean modal)
        {
            this(parent, modal);
            setTitle(title);
        }

    /**
     * Shows or hides the component depending on the boolean flag b.
     * @param b  if true, show the component; otherwise, hide the component.
     * @see java.awt.Component#isVisible
     */
    public void setVisible(boolean b)
        {
            if(b)
            {
                Rectangle bounds = getParent().getBounds();
                Rectangle abounds = getBounds();
                setLocation(bounds.x + (bounds.width - abounds.width)/ 2,
                        bounds.y + (bounds.height - abounds.height)/2);
            }
            super.setVisible(b);
        }

    // Used for addNotify check.
        boolean fComponentsAdjusted = false;

        //DECLARE_CONTROLS
        java.awt.Button yesButton;
        java.awt.Button noButton;
        java.awt.Label label1;
        //
        class SymWindow extends java.awt.event.WindowAdapter
        {
            public void windowClosing(java.awt.event.WindowEvent event)
            {
                Object object = event.getSource();
                if (object == QuitDialog.this)
                    QuitDialog_WindowClosing(event);
            }
```

```
    }

    void QuitDialog_WindowClosing(java.awt.event.WindowEvent event)
    {
    dispose();
    }

    class SymAction implements java.awt.event.ActionListener
    {
        public void actionPerformed(java.awt.event.ActionEvent event)
        {
            Object object = event.getSource();
            if (object == noButton)
                noButton_Clicked(event);
            else if (object == yesButton)
                yesButton_Clicked(event);
        }
    }

    void yesButton_Clicked(java.awt.event.ActionEvent event)
    {
        Toolkit.getDefaultToolkit().getSystemEventQueue().postEvent(new
            WindowEvent((java.awt.Window)getParent(),
        WindowEvent.WINDOW_CLOSING));
    }

    void noButton_Clicked(java.awt.event.ActionEvent event)
    {
      dispose();
    }

}

class BackPropagation {

    static float squash(float input)
    // squashing function
    // use sigmoid -- can customize to something
    // else if desired; can add a bias term too
    //
    {
        if (input < -50)
            return (float)0.0;
        else    if (input > 50)
            return (float)1.0;
        else return (float)(1/(1+Math.exp(-(double)input)));
```

```
        }

        static float randomweight(int init)
        {
                // random number generator
                // will return a floating point
                // value between -1 and 1
                return (float)(2*Math.random()-1);
        }

}

class BackPropagation {

        static float squash(float input)
        // squashing function
        // use sigmoid -- can customize to something
        // else if desired; can add a bias term too
        //
        {
                if (input < -50)
                        return (float)0.0;
                else    if (input > 50)
                        return (float)1.0;
                else return (float)(1/(1+Math.exp(-(double)input)));
        }

        static float randomweight(int init)
        {
                // random number generator
                // will return a floating point
                // value between -1 and 1
                return (float)(2*Math.random()-1);
        }

}

import java.io.*;
class Controller {

float error_tolerance=0.1;
float total_error=0.0;
float avg_error_per_cycle=0.0;
float error_last_cycle=0.0;
float avgerr_per_pattern=0.0; // for the latest cycle
float error_last_pattern=0.0;
```

```
float learning_parameter=0.02;
unsigned temp, startup;
long vectors_in_buffer;
long max_cycles;
long patterns_per_cycle=0;

long total_cycles, total_patterns;
int i;


network backp;
String str=null;


void train()
{
// open output file for writing
try{
        trdout=new DataOutputStream(new FileOutputStream(OUPUT_FILE,"w"));
}catch(Exception e){
        System.out.println("Error at"+OUTPUT_FILE+e);
        System.exit(1);
}
// enter the training mode : 1=training on     0=training off
System.out.println ( "-------------------------------------------------");
System.out.println ( " C++ Neural Networks and Fuzzy Logic ");
System.out.println ( " Backpropagation simulator ");
System.out.println ( "            version 1 ");
System.out.println ( "-------------------------------------------------");
System.out.println ("Please enter 1 for TRAINING on, or 0 for off: \n");
System.out.println ("Use training to change weights according to your");
System.out.println ("expected outputs. Your training.dat file should contain");
System.out.println ("a set of inputs and expected outputs. The number of");
System.out.println ("inputs determines the size of the first (input) layer");
System.out.println ( "while the number of outputs determines the size of the");
System.out.println ( "last (output) layer :\n");

cin >> temp;
backp.set_training(temp);

if (backp.get_training_value() == 1)
        {
        System.out.println("--> Training mode is *ON*. weights will be saved");
        System.out.println("in the file weights.dat at the end of the");
        System.out.println("current set of input (training) data");
        }
```

```
else
        {
        System.out.println("--> Training mode is *OFF*. weights will be loaded");
        System.out.println("from the file weights.dat and the current");
        System.out.println("(test) data set will be used. For the test");
        System.out.println("data set, the test.dat file should contain");
        System.out.println("only inputs, and no expected outputs.");
        }

if (backp.get_training_value()==1)
        {
        // ---------------------------------------
        //      Read in values for the error_tolerance,
        //      and the learning_parameter
        // ---------------------------------------
        System.out.println(" Please enter in the error_tolerance");
        System.out.println(" --- between 0.001 to 100.0, try 0.1 to start --");
        System.out.println();
        System.out.println("and the learning_parameter, beta");
        System.out.println(" --- between 0.01 to 1.0, try 0.5 to start -- \n");
        System.out.println(" separate entries by a space");
        System.out.println(" example: 0.1 0.5 sets defaults mentioned :\n");

        try{
                str=sysdin.readLine();
                error_tolearance=Float.valueOf(str).floatValue();
                str=sysdin.readLine();
                learning_parameter=Float.valueOf(str).floatValue();
        }catch(Exception e){
                System.out.println("Error in reading"+e);
                System.exit(1);
        }
        //---------------------------------------
        // open training file for reading
        //---------------------------------------
        try{
                tdin=new DataInputStream(new FileInputStream(TRAING_FILE));
        }catch(Exception e){
                System.out.println("Error in opneing the file "+e);
                System.exit(1);
        }
        ddin=tdin;
        // Read in the maximum number of cycles
        // each pass through the input data file is a cycle
        System.out.println("Please enter the maximum cycles for the simulation");
        System.out.println("A cycle is one pass through the data set.");
```

```
        System.out.println( "Try a value of 10 to start with");
        try{
                str=sysdin.readLine();
                max_cycles=Float.valueOf(str).floatValue();
        }catch(Exception e) {
                System.out.println("Error in reading the maxcycles'+e);
                System.exit(1)_;
        }

        }
else
        {
        try{
                tdin=new DataInputStream=new DataInputStream(ne FileInputStream(
                        TEST_FILE));
        }catch(Exception e){
        System.out.println("Problem in opening the file ");
        System.exit(1);
        }
        ddin=tdin;
        }



// the main loop
//
// training: continue looping until the total error is less than
//              the tolerance specified, or the maximum number of
//              cycles is exceeded; use both the forward signal propagation
//              and the backward error propagation phases. If the error
//              tolerance criteria is satisfied, save the weights in a file.
// no training: just proceed through the input data set once in the
//              forward signal propagation phase only. Read the starting
//              weights from a file.
// in both cases report the outputs on the screen


// intialize counters
total_cycles=0; // a cycle is once through all the input data
total_patterns=0; // a pattern is one entry in the input data



// get layer information
backp.get_layer_info();
```

```
// set up the network connections
backp.set_up_network();

// initialize the weights
if (backp.get_training_value()==1)
        {
        // randomize weights for all layers; there is no
        // weight matrix associated with the input layer
        // weight file will be written after processing
        // so open for writing
        try{
                wdout=new DataOutputStream(new
FileOutputStream(WEIGHTS_FILE));
        }catch(Exception e){
                System.out.println("problem in opening the output file ");
                System.exit(1);
        }
        backp.randomize_weights();
        }
else
        {
        // read in the weight matrix defined by a
        // prior run of the backpropagation simulator
        // with training on
        try{
                wdin=new DataInputStream(new FileInputStream(WEIGHTS_FILE));
        }catch(Exception e) {
                System.out.println("Error in opening the file "+e);
                System.exit(1);
        }
        backp.read_weights(weights_file_ptr);
        }



// main loop
// if training is on, keep going through the input data
//              until the error is acceptable or the maximum number of cycles
//              is exceeded.
// if training is off, go through the input data once. report outputs
// with inputs to file output.dat

startup=1;
vectors_in_buffer = MAX_VECTORS; // startup condition
```

```
total_error = 0;


while (                 ((backp.get_training_value()==1)
                        && (avgerr_per_pattern
                                > error_tolerance)
                        && (total_cycles < max_cycles)
                        && (vectors_in_buffer !=0))
                        || ((backp.get_training_value()==0)
                        && (total_cycles < 1))
                        || ((backp.get_training_value()==1)
                        && (startup==1))
                        )
{
startup=0;
error_last_cycle=0; // reset for each cycle
patterns_per_cycle=0;
// process all the vectors in the datafile
// going through one buffer at a time
// pattern by pattern



while ((vectors_in_buffer==MAX_VECTORS))
        {

        vectors_in_buffer=
                backp.fill_IObuffer(data_file_ptr); // fill buffer
                if (vectors_in_buffer < 0)
                        {
                        System.out.println ("error in reading in vectors, aborting");
                        System.out.println ("check that there are on extra linefeeds");
                        System.out.println ("in your data file, and that the number");
                        System.out.println ("of layers and size of layers match the");
                        System.out.println ("the parameters provided.");
                        System.exit(1);
                        }

                // process vectors
                for (i=0; i<vectors_in_buffer; i++)
                        {
                        // get next pattern
                        backp.set_up_pattern(i);

                        total_patterns++;
                        patterns_per_cycle++;
```

```
                    // forward propagate

                    backp.forward_prop();

                    if (backp.get_training_value()==0)
                            backp.write_outputs(output_file_ptr);

                    // back_propagate, if appropriate
                    if (backp.get_training_value()==1)
                            {

                            backp.backward_prop(error_last_pattern);
                            error_last_cycle +=
                                    error_last_pattern*error_last_pattern;
                            backp.update_weights(learning_parameter);
                            // backp.list_weights(); // can
                            // see change in weights by
                            // using list_weights before and
                            // after back_propagation
                            }

                    }

            error_last_pattern = 0;
            }

    avgerr_per_pattern=((float)Math.sqrt((double)error_last_cycle/patterns_per_cycle));
    total_error += error_last_cycle;
    total_cycles++;

    // most character displays are 25 lines
    // user will see a corner display of the cycle count
    // as it changes

    System.out.println ( "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
    System.out.println ( total_cycles +"\t" +" avgerr_per_pattern" );

    fseek(data_file_ptr, 0L, SEEK_SET); // reset the file pointer
                                        // to the beginning of
                                        // the file
    vectors_in_buffer = MAX_VECTORS; // reset

    } // end main loop

    System.out.println ( "\n\n\n\n\n\n\n\n\n\n\n");
    System.out.println ( "-------------------------------------------------");
```

```
System.out.println ( " done:   results in file output.dat");
System.out.println ( "         training: last vector only");
System.out.println ( "         not training: full cycle\n");
if (backp.get_training_value()==1)
        {
        backp.write_weights(weights_file_ptr);
        backp.write_outputs(output_file_ptr);
        avg_error_per_cycle= (float)Math.sqrt((double)total_error/total_cycles);
        error_last_cycle=(float)Math.sqrt((double)error_last_cycle);

System.out.println("          weights saved in file weights.dat");
System.out.println() ;
System.out.println( "---->average error per cycle = " + avg_error_per_cycle + " <---");
System.out.println( "---->error last cycle = " + error_last_cycle + " <---");
System.out.println( "->error last cycle per pattern= " + avgerr_per_pattern + " <---");

        }

System.out.println("---------->total cycles = " + total_cycles +"---");
System.out.println("---------->total patterns = " +total_patterns+ " +---");
System.out.println("-------------------------------------------------");
// close all files

}
}

import java.util.*;
import java.io.*;
/*//////////////////////////////////////////////////////////////////////
Agent manager creates agents and provides the references of all agents .
Agent Manager also assign the tasks to agents according to the gaussian
distribution
//////////////////////////////////////////////////////////////////////*/

class AgentManager extends Thread implements Constants {
    static Agents agentThread[];   // agent references list
    static int totalNumOfAgents;   // total number of agents in the list
    static int totalTasks=0;       // total tasks submitted to the system
    GausRandom random;             // gaussian random varaible
    int scheduledTasks;            // scheduled Tasks to be executed
    PrintWriter out=null;          // log file
    Network net=null;              // network reference
      long startTime;
    ThreadGroup agentThreadGroup=null;// thread group;
      static int totalTime=1000;
    // constructor of agent manager
```

```
        AgentManager(Network net,PrintWriter out,int numOfAgents,boolean
isCloned,int num)
        {
                this.net=net;
                this.out=out;
                scheduledTasks=num;
                random=new GausRandom(77889765);
                totalNumOfAgents=numOfAgents;
                Agents.isClonable=isCloned;
                agentThreadGroup=new ThreadGroup("Agent Cloning");
                System.out.println("Agent Manager Created...");
        }
      // This method will create the agent threads intially
      // and start the threads
        void createAgents()
        {
                System.out.println("Creating the Agents ...");
                agentThread=new Agents[totalNumOfAgents];
                for(int i=0;i<agentThread.length;i++)
                        agentThread[i]=new Agents(i,net,out,agentThreadGroup);
                for(int i=0;i<agentThread.length;i++)
                        agentThread[i].start();
                System.out.println("Agents created ...\n");
        }
      // adds the newly created cloned thread to reference list
        static synchronized void increaseAgents(Agents cloneThread)
        {
        Agents[] temp;
          if(totalNumOfAgents== agentThread.length) { // check for array size
                        temp=new Agents[2*agentThread.length];
                        for(int i=0;i<agentThread.length;i++)
                                temp[i] = agentThread[i];
                        temp[agentThread.length]=cloneThread;
                        agentThread=temp;
                }else
                        agentThread[totalNumOfAgents]=cloneThread;
                totalNumOfAgents++;
        }
      // assigns the tasks to agents according to the gaussion distribution
        void assignTasks(int numOfTasks)
        {
        int agentNum;
        int numTasks;
        Task task[];
                for(int i=0;i<totalNumOfAgents && totalTasks<=numOfTasks;i++){
                        agentNum=random.uniformRandom(totalNumOfAgents);
```

```
                numTasks=random.gausRandom(10);
                totalTasks+=numTasks;
                task=new Task[numTasks];
                for(int j=0;j<numTasks;j++)
                        task[j]=new Task();
                agentThread[agentNum].assignTasks(task);
            }
        }
    // controling method of agent manager thread
        public void run()
        {
        int i;
        int time;
        int numOfAgents;
        Runtime runtime=Runtime.getRuntime();
            numOfAgents=totalNumOfAgents;
            for(i=scheduledTasks;i<=scheduledTasks;i+=100){
                time = totalTime*1000/i;
                startTime=System.currentTimeMillis();
                totalTasks=0;
                Agents.completedTasks=0;
                totalNumOfAgents=numOfAgents;
                createAgents();
                while(totalTasks<i) {
                        try{
                                Thread.sleep(time);
                        }catch(InterruptedException e) {
                                System.out.println(e);
                        }
                        assignTasks(i);
                }
                out.println("total Tasks "+ totalTasks);
                out.println("Completed Tasks"+Agents.completedTasks);
                for(int j=0;j<totalNumOfAgents;j++)
                        agentThread[j].stopJobs();
                for(int j=0;j<totalNumOfAgents;j++)
                        agentThread[j].releaseResources();
//              for(int j=numOfAgents;j<totalNumOfAgents;j++)
//                      if(agentThread[j].isAlive())
//                              agentThread[j].stop();
                System.out.println("Proceeding for the "+ i +" Tasks");
                System.out.println("Total Tasks "+ totalTasks);
                System.out.println("Completed Tasks "+
Agents.completedTasks);
                agentThreadGroup.stop();
            }
```

```
//              for(int j=0;j<totalNumOfAgents;j++)
//                      agentThread[j].stop();
//              agentThreadGroup.stop();
                stop();
        }


}

//                                      output layer
//-----------------------------------------
import java.io.*;

class OutputLayer extends Layer{

        protected float[] weights;
        protected float[] output_errors; // array of errors at output
        protected float[] back_errors; // array of errors back-propagated
        protected float[] expected_values;      // to inputs

        public OutputLayer(int in, int out)
        {
                num_inputs=in;
                num_outputs=out;
                weights=new float[num_inputs* num_outputs];
                output_errors=new float[num_outputs];
                back_errors=new float[num_inputs];
                outputs = new float[num_outputs];
                expected_values = new float[num_outputs];
        }

        public void calc_out()
        {
        int i,j,k;
        float accumulator=0.0f;

                for (j=0; j<num_outputs; j++){
                        for (i=0; i<num_inputs; i++){
                        k=i*num_outputs;
                        if (weights[k+j]*weights[k+j] > 1000000.0)
                        {
                                System.out.println( "weights are blowing up");
                                System.out.println("try a smaller learning constant");
                                System.out.println( "e.g. beta=0.02    aborting...");
        //                      System.exit(1);
                        }
```

```
                outputs[j]=weights[k+j]* inputs[i] ;
                accumulator+=outputs[j];
                }
                // use the sigmoid squash function
                outputs[j]=BackPropagation.squash(accumulator);
                accumulator=0;
        }
    }

// calculates the error
    public float calc_error(float error)
    {
    int i, j, k;
    float accumulator=0;
    float total_error=0;

        for (j=0; j<num_outputs; j++)
        {
                output_errors[j] = expected_values[j]-outputs[j];
                total_error+=output_errors[j];
        }
        error=total_error;
        for (i=0; i<num_inputs; i++)
        {
                k=i*num_outputs;
                for (j=0; j<num_outputs; j++)
                {
                        back_errors[i]=         weights[k+j]*output_errors[j];
                        accumulator+=back_errors[i];
                }
                back_errors[i]=accumulator;
                accumulator=0;
        // now multiply by derivative of
        // sigmoid squashing function, which is
        // just the input*(1-input)
        back_errors[i]*=inputs[i] *(1-inputs[i]);
        }
    return error;
    }
// randomize the weights of the network
    public void randomize_weights()
    {
    int i, j, k;
    final int first_time=1;
    final int not_first_time=0;
    float discard;
```

```
            discard=BackPropagation.randomweight(first_time);
            for (i=0; i< num_inputs; i++)
            {
                    k=i*num_outputs;
                    for (j=0; j< num_outputs; j++)

    weights[k+j]=BackPropagation.randomweight(not_first_time);
            }
    }

    public void update_weights(final float beta)
    {
    int i, j, k;

    // learning law: weight_change =
    //              beta*output_error*input
            for (i=0; i< num_inputs; i++)
            {
                    k=i*num_outputs;
                    for (j=0; j< num_outputs; j++)
                    weights[k+j] +=
                    beta*output_errors[j]*inputs[i];
            }

    }
// list weights
    public void list_weights()
    {
    int i, j, k;

            for (i=0; i< num_inputs; i++)
            {
                    k=i*num_outputs;
                    for (j=0; j< num_outputs; j++)
                            System.out.println("weight["+i+","+j+"] is: "+
                weights[k+j]);
            }
    }
// lists the errors
    public void  list_errors()
    {
    int i, j;

            for (i=0; i< num_inputs; i++)
```

```
                System.out.println("backerror["+i+"] is : "+back_errors[i]);

        for (j=0; j< num_outputs; j++)
                System.out.println("outputerrors["+j+"] is: "+output_errors[j]);
}


void write_weights(int layer_no,DataOutputStream  out)
{
int i, j, k;

// assume file is already open and ready for
// writing
// prepend the layer_no to all lines of data
// format:
//              layer_no        weight[0,0] weight[0,1] ...
//              layer_no        weight[1,0] weight[1,1] ...
//              ...
        try{
        for (i=0; i< num_inputs; i++)
        {
                out.writeInt(layer_no);
                out.writeChar('\n');
                k=i*num_outputs;
                for (j=0; j< num_outputs; j++){
                        out.writeFloat(weights[k+j]);
                        out.writeChar('\n');
                }
        }
        }catch(Exception e){}
}
// read the weights
public void  read_weights(int layer_no,DataInputStream  din)
{
int i, j, k;

// assume file is already open and ready for
// reading
// look for the prepended layer_no
// format:
//              layer_no        weight[0,0] weight[0,1] ...
//              layer_no        weight[1,0] weight[1,1] ...
//              ...
        try{
        while (true)
   {
```

```
                    j=din.readInt();
                    din.readChar();
                    if (j==layer_no)
                            break;
                    else
                    {
                            for(int p=0;p<num_outputs;p++){
                                    din.readFloat();
                                    din.readChar();
                            }
                    }
            }

            // continue getting first line
            i=0;
            for (j=0; j< num_outputs; j++){
                    weights[j]=din.readFloat();
                    din.readChar();
            }
        // now get the other lines
            for (i=1; i< num_inputs; i++)
            {
                    layer_no=din.readInt();
                    din.readChar();
                    k=i*num_outputs;
                    for (j=0; j< num_outputs; j++)
                    {
                            weights[k+j]=din.readFloat();
                            din.readChar();
            }

            }
        }catch(Exception e) {}
    }
    // lists the outputs
      public void list_outputs()
      {
      int j;

            for (j=0; j< num_outputs; j++)
            {
            System.out.println("outputs["+j+"] is: "+outputs[j]);
            }
      }
}
```

```
import java.util.*;
import java.io.*;
// this class creates the network
class Network implements Constants{
    int spDist[][];       //shortest path from one node to another
    int pred[][];         //predecessor matrix
    Host hostServer[];    // reference to all servers
    GausRandom random;    // gaussian random varaible
    PrintWriter out=null; // log file reference
    static int numOfHosts; // number of hosts
    // constructor of network
      Network(PrintWriter out,int numOfHosts)
      {
      int x,y;
            this.out=out;
            random=new GausRandom(688888885);
            hostServer=new Host[numOfHosts];
            this.numOfHosts=numOfHosts;
            Host.numOfHosts=numOfHosts;
            for(int i=0;i<numOfHosts;i++) {
                  do {
                        x=random.uniformRandom(MAXX);
                        y=random.uniformRandom(MAXY);
                  }while(checkForPosition(i,x,y));
                  hostServer[i]=new Host(i,x,y);
            }
        spDist=new int[numOfHosts][numOfHosts];
            pred=new int[numOfHosts][numOfHosts];
      }
    // checks for the position concidence with other host
      boolean checkForPosition(int index,int x,int y)
      {
      boolean done=false;
            for(int i=0;i<index&& !done ; i++)
                  if( Math.abs(hostServer[i].x -x )<10 &&
                        Math.abs(hostServer[i].y-y) <10)
                        done=true;
            return done;
      }
    // creates the network and find out shortest path
      public void createNetwork()
      {
            floydRouting();
            for(int i=0;i<numOfHosts;i++)
```

```
                    setRoutes(i);
                showOutput();
        }
    // checks for connection between two nodes
        boolean isConnected(int i,int j)
        {
        boolean done=false;
                for(int k=0;k<hostServer[i].connection.length  && !done ; k++)
                        if(hostServer[i].connection[k] == j)
                    done=true;
                return done;
        }
    // find the distance between two nodes
        int findDist(int i,int j)
        {
        double sum=0;
                if(isConnected(i,j)) {
                        sum=(hostServer[i].x-hostServer[j].x) *
                                (hostServer[i].x- hostServer[j].x);
                        sum+=(hostServer[i].y-hostServer[j].y) *
                                (hostServer[i].y -hostServer[j].y);
                        return (int) Math.sqrt(sum);
                }else
                        return INFINITY;
        }
    // finds the shortest path in the network using floyd algorithm
        public void floydRouting()
        {
                for(int i=0;i<numOfHosts;i++)
                for(int j=0;j<numOfHosts;j++){
                        if(i!=j)
                                spDist[i][j]=findDist(i,j);
                        else
                                spDist[i][i]=0;
                        pred[i][j]=i;
                }
                for(int k=0;k<numOfHosts;k++)
                for(int i=0;i<numOfHosts;i++)
                for(int j=0;j<numOfHosts;j++)
                if(spDist[i][j]> spDist[i][k]+spDist[k][j]) {
                        spDist[i][j]=spDist[i][k]+spDist[k][j];
                        pred[i][j]=pred[k][j];
                }
        }
    // set the routes of the id
        void setRoutes(int id)
```

```
        {
                for(int i=0;i<numOfHosts;i++){
                        hostServer[id].dist[i].distance=spDist[id][i];
                        hostServer[id].dist[i].pred=pred[id][i];
                }
        }
    // shows the output path of a server
        void showOutput()
        {
                System.out.println("Showing the output ");
                for(int i=0;i<numOfHosts;i++){
                for(int j=0;j<numOfHosts;j++)
                        System.out.print(spDist[i][j] + " "+pred[i][j]+ " ");
                System.out.println();
                }
        }
}
```