A Distributed Multi Agent Systems Framework

Authors: Manish Jain, I.V. Aprameya Rao

Supervisor: Prof. Kamalakar Karlapalem

International Institute of Information Technology, Hyderabad

2006

ABSTRACT

Building multi-agent systems that can scale up to very large number of agents is a challenging research problem. In this project, we have developed Distributed Multi Agent System Framework (DMASF), a system which can simulate billions of agents in thousands of seconds. DMASF is a generic and versatile tool that can be used for building massive multi agent system applications. DMASF utilizes distributed computation to gain performance as well as a database to manage the agent and environment state. We discuss the design and implementation of DMASF in detail and present experimental results.

Table of Contents

1. Introduction

Many Multi Agent Simulation Systems do not scale to a large number of agents. With dropping hardware costs, computer networks are present almost everywhere. Many of the Multi Agent Systems do not utilize the advantages of distributing the simulation work across multiple computers in a networked environment.

The ability to pause and resume complex simulations is something that is missing in most Multi Agent System simulators. This applies more to simulators that use a main memory based simulation model (in which different threads are used for executing different agents).

Most systems do not allow defining changes in the environment. For example, if the simulator has only 2D visualizations then it is not possible to add a 3D view to it. They come with their own interpreted language that has a steep learning curve. They are also not powerful enough for expressing common programming constructs. For example, though the interpreted language of NetLogo [5] makes it easier for non-programmers to use the system, the lack of advanced data structures could hamper the use of NetLogo in complex scenarios.

The simulators that employ distributed computing are difficult to set up and maintain. There is no straightforward method of installing and deploying them. The time taken to build and deploy a simulation over a distributed system is considerably high. DMASF overcomes these limitations.

DMASF (Distributed Multi Agents Simulation Framework) is written in Python [9]. It has been designed to distribute computational tasks over multiple computers and thereby simulate a large number (up to billions) of agents. DMASF is available for general use. In DMASF, an agent is an autonomous entity working towards a goal. A Multi Agent System can be defined as a group of agents working together to achieve their goals.

The DMASF core is small and lightweight. It has been designed such that a user can develop simulations and applications in it without any difficulty and an experienced developer can extend the core and build domain specific applications. The DMASF GUI currently provides a 2D representation of the world. It can be directly extended by the developer to create 3D visualizations of the simulation.

DMASF uses a database for storing agent state information. This helps DMASF in scaling to a large number of agents. It can also be run without the GUI for simulations in which speed is the paramount criterion, animation of agent behavior is not important and cumulative results are sufficient.

In this paper we will discuss the design of DMASF as well as some of the results we have obtained by running various test simulations to illustrate the scalability of DMASF. We will also describe some applications scenarios where DMASF has been used.

2. Motivation and Design Goals

The traditional paradigm for Multi Agent Systems is to run each agent in a separate thread. This clearly represents the agents being concurrent autonomous entities. However, this mapping of the agent world to the computer world does not allow for execution of a large number of agents. On a standard desktop computer system (2GHz Processor, 512 MB main memory) we cannot go beyond 1000 threads.

Therefore, we need a different model. The model of execution used by DMASF is similar to the model used by current Operating Systems to run multiple tasks at the same time [10]. Each task is given a time-slice of CPU time. It is then suspended and another task is run. We schedule and execute agents in a similar manner. Each agent has an update function that specifies what the agent does. The update functions for all agents are run over and over in an iterative manner till the simulation ends.

Distributed computing by default is much cheaper than the alternative of doing all the computation on one very powerful machine. DMASF employs distributed computing to obtain high scalability. To keep the hardware requirements of the individual computer systems low, we have designed and built DMASF to have a very small footprint.

This scale up or ability to simulate a very large number of agents helps in increasing the number of domains and scope of application of Multi Agent System simulation techniques. It also enables more fine grained simulation. For example, in a RoboRescue [12] environment it is feasible to simulate 100,000 civilians and their behavior using DMASF.

In a Multi Agent Simulation, especially one that is distributed, it is imperative that the simulator provide the ability to pause and resume the simulation because the required computers may not be available all the time. DMASF allows the user to pause and resume the simulation after each iteration.

Different Multi Agent Systems can have different ways of visualizing the world. Since we did not wish to constrain the developer to a system defined world view, we built an extensible GUI. The developer can not only define what his agents do, but what they look like as well.

The contribution of this paper is in to build a new system with the following design goals:

- A fast lightweight core.
- Distributed computing for high performance.
- Scalability for hundreds of millions of agents without the GUI.
- A separate extensible GUI.
- Saving the simulation state so that it could be resumed at any time.
- Easy extensibility.

3. Design Issues and Challenges

An agent in DMASF is represented by an agent type and an agent id. Each agent type can have an independent set of properties. Multiple agent types and multiple agents of a single type are permitted.

An agent cannot change its type directly. To implement type changing, the environment would need to kill this agent and create an agent of the new type. In DMASF an agent lives until it is explicitly killed by the environment or the simulation ends.

A host is an instance of DMASF running on a computer. A computer can run multiple hosts at the same time. Our model has a set of simulator objects on each host

We use a database for storing agent state information. Databases already have excellent query mechanisms and are very robust. Databases can easily store and retrieve information for billions of tuples and thus help in achieving impressive scale ups. To illustrate this, let us consider 100 million agents. If we use a single integer to represent each agent, it will require 380 Megabytes. This cannot be kept on a computer with 256 Megabytes of main memory. In DMASF, a fixed number of agents are kept in main memory at a time while the rest are flushed to a database system so that retrieval and update is performed efficiently.

Also, we need to provide the same view of the world at each iteration to all the agents. We hence cannot commit the updates made by an agent to the database immediately as the simulation would then present two world views. One has to wait until all agents have finished updating. Again, due to the sheer number of such updates we cannot`n`
`Mozilla Firefox 3` store these updates in main memory. We thus keep a fixed number of updates in main memory and write the other updates to secondary storage. DMASF has a setting to override this default behavior if the simulation requires updates to be visible immediately.

Another challenge in implementing a distributed computational system is to schedule or decide which agents should be simulated on which machines. We cannot allocate an equal number of agents to each host as slower hosts will then tend to slow down the faster ones. Therefore, DMASF has a dynamic scheduler that assesses the performance of each machine and dynamically schedules and load balances agents on them. This scheduler will be described in detail later.

The hosts in DMASF are organized in a client-server architecture. The server decides which agents are to be simulated on which host. It is also responsible for the synchronization among hosts.

Query results that give common world states are cached so that the simulation runs faster. In a simulation in which agents had to move in a circle, this caching reduced simulation time from O(n2) to O(n) where n is the number of agents. Whenever an agent requests such information, it is provided from the cache instead of running the query. This reduces the number of database queries, the load on the database system as well as avoids redundant computation.

4. The Distributed Multi Agent System Framework

The main components that define a Multi Agent System are (1) Agent Type Definition, (2) Agent Behavior Definition, (3) Agent to Agent Communication, (4) Environmental Definition and, (5) GUI.

4.1.Agent Type Definition

An agent type is defined by a call to the *RegisterAgentType* function which will tell DMASF the properties and the *update* function for that type. Each agent type can have a distinct set of properties. Properties can be of the following primitive data types: (1) Integer, (2) Float, (3) String and, (4) Large String. Complex Python data types can be represented using the large string data type and the Python pickle module (which converts any data type to a string). Agents have some default properties such as *id* (identifier of the agent), *size* (defines how big the agent would look in the GUI), *x, y* (specifies the co-ordinates of the agent in the world).

Example: A world with three types of agents - *helicopter*, *smoke* and *rescue vehicles*. The environment would spawn *smoke* agents randomly representing new fires in the area. The *helicopter* agents can see the entire world and are responsible for informing the *rescue vehicle* agents about the various *smoke* agents. Here, the *rescue vehicle* agents would have properties representing which *smoke* agent it is going to tackle. The *smoke* agents would have a property that defines how many more iterations until the *smoke* agent dies (i.e. the fire burns out on its own).

4.2.Agent Behavior Definition

The agent behavior is defined using a class. The user needs to extend the default-agent-handler class that is provided with DMASF. The class already has some built-in functions and properties such as:

- ***writeState***: writes the modified properties to the data store.
- ***writeStateSynchronous***: writes the modified properties to the data store immediately. These changes are now visible to the next agent in the world view.
- ***sendMessage***: to send a message to another agent.
- ***getMessages***: to retrieve the messages that have been sent to the agent.
- ***kill***: to kill the agent.
- ***fields***: Python dictionary that provides the values of the properties of the agent.
- ***db***: a low level interface to the database for writing custom SQL queries.

Moreover, the user should overload the update function with the agent behavior. This is the code that is executed by DMASF whenever the agent is scheduled.

We have also provided some simple aggregate functions to help executing common database queries. These can be used for implementing various percepts of the agent. In case even finer control is required, the developer can write SQL queries to directly access the database.

Example (continued): The *rescue vehicle* agents would contain code that moves the agent in the world towards the *smoke* agents as well as code for extinguishing the fire. The *helicopter* agents would contain SQL queries for scanning the world for new *smoke* agents and then informing the *rescue vehicle* agents about them. The *smoke* agents do nothing other than waiting to be either extinguished by the *rescue vehicles* or die out on their own.

4.3.Agent to Agent Communication

Messages to the agent are represented by a *from_address* (specifying the type and identifier of the agent who sent the message), a *simtime* (the iteration number at which the message was sent) and the message (a string specifying the contents of the message). Agents can access these messages using the *getMessages* function. They can send messages using the *sendMessage* function. There is no limit on the number of messages an agent can receive. By default, messages are flushed after each iteration. This is done to improve the performance. Messages are stored in the database and transferred to the *host* that is executing the agent code over TCP/IP.

Example (continued): The communication between the agents happens when the *helicopter* agents inform the *rescue vehicle* agents about new *smoke* agents (fires).

4.4.Environmental Definition

Just as the agent behavior is modeled by a class, the environment behavior is also modeled by a class. The user needs to extend the default-world-handler class. The user can overload the following functions:

- ***begin***: This function is called at the beginning of the simulation. The user can setup the environment here. Typically, this function will contain code for setting up the world as well as creating the initial agents.
- ***beginUpdate***: This function is called at the beginning of each iteration. The agent code is executed only after the execution of this function finishes.
- ***endUpdate***: This function is called at the end of each iteration. An iteration is assumed to have completed only after this function finishes execution. If this function returns *True* then DMASF assumes that the simulation's goal has been reached. If it returns *False* the simulation will continue to run.
- ***end***: This function is called when the simulation ends. The simulation ends when it has run for the specified number of iterations or by exiting DMASF or the *endUpdate* function returns *True*.

Example (continued): The environment would be primarily responsible for creating new *smoke* agents (fires). This would be done in the *beginUpdate* function. Depending on the iteration number (time), *smoke* agents would be spawned with appropriate properties.

4.5.Framework wide options

DMASF has some built-in functions to enhance the scope of the simulation as well as tweaking system wide properties.

- ***setMaxSimTime***: This specifies the number of iterations for which the simulation should run. By default the simulation will run for infinite number of iterations or until the *endUpdate* function of the world handler returns *True*.
- ***disableFlushingMessages***: By default DMASF will flush messages (clear inbox) for each agent after the end of each iteration. However, if a simulation requires messages to be stored then this function can be used to override the default behavior.
- ***setSynchronousWrites***: By default when an agent writes his state, it is put inside a temporary store. When all the agents have finished updating, all agent updates are then committed at once. This is done to present the same world view to each agent during an iteration. However, if it is imperative for agent updates to be visible immediately then this mode should be enabled. The agent should also use *writeStateSynchronous* instead of *writeState*. If this mode is enabled, DMASF will also not cache any values of aggregate functions.
- ***setDrawAllFlag***: By default the GUI will draw all the agents in the world. This feature can be disabled using this function. The user should then specify which agents he would like to display using the *addDrawAgent* function.

Example (continued): In this simulation, the maximum *simtime* was set to -1 (infinity) so that the simulation would go on until explicitly killed. Flushing of messages was enabled because we did not need to store messages. Synchronous writes were also disabled because during an iteration, we wished to present the same world view to all the agents.

4.6.Graphical Interface

The GUI has been developed in OpenGL. The user can write OpenGL code specifying how the agent should be drawn. While defining the type of the agent, the user can provide a render function that will be called whenever the GUI needs to draw an agent of that type. The GUI can also be configured to draw only specific agents. This is useful for observing the behavior of only a few specific agents.

Example (continued): We provided DMASF with custom render functions to draw the three types of agents. *Rescue Vehicles* were drawn as a convex polygon in the shape of a car. *Helicopter* agents were drawn in the same fashion. *Smoke* agents were drawn by four overlapping circles.

4.7.Developing and Deploying Simulations

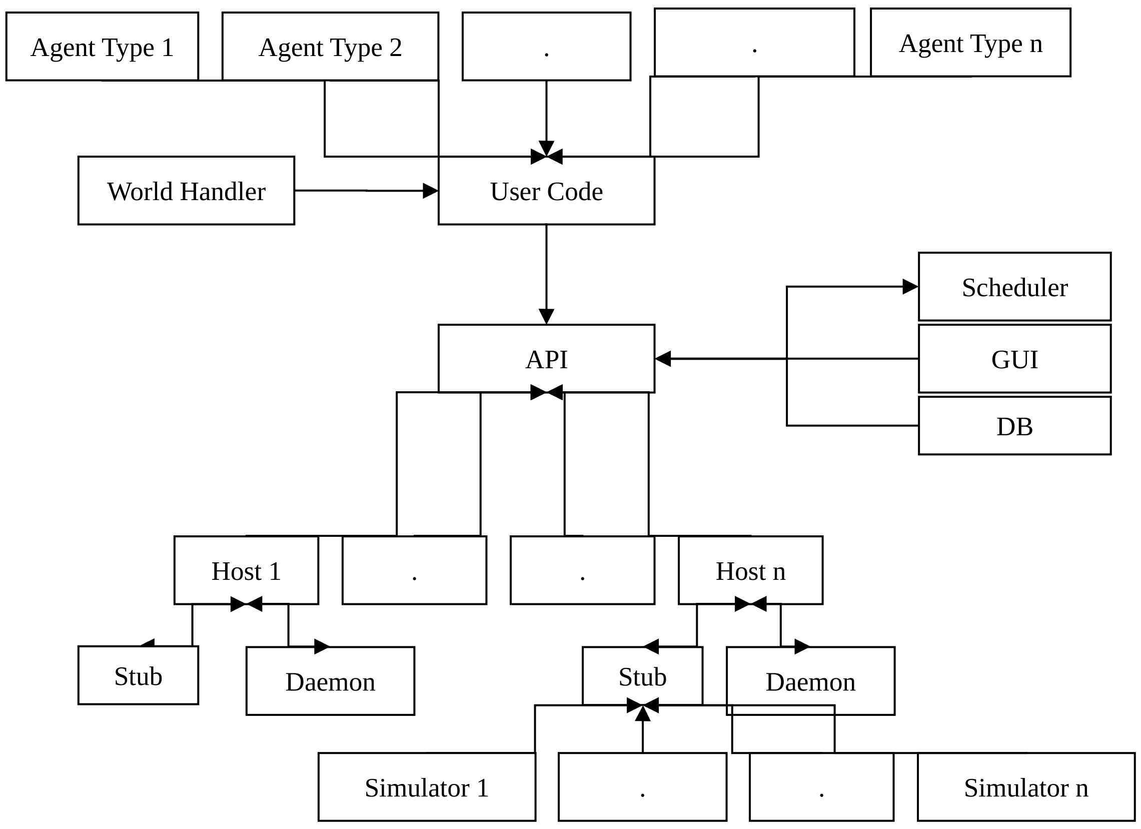
Developing a simple simulation consists of first defining the agent type. Next, the user needs to define the behavior of that type. Once the simulation has been developed it needs to be run on different computer systems. Adding a computer system consists of running the *apiclient* on that system and telling DMASF to use that particular system as a *host*. To run the *apiclient*, the user needs to start the daemon. The user code will automatically be transferred across the network and will be ready for execution.

4.8.Hardware Requirements

DMASF does not require any special hardware to run. We have tested DMASF on machines with 256MB RAM and 1.7 GHz AMD processors. It3 has also been tested on a computer system with four 3GHz processors and 2GB of main memory. There are settings to allow the user to control the number of agents that should be kept in main memory thereby controlling the amount of memory used by DMASF.

Example (continued): This simulation was run on a single *host* with four 3GHz processors and 2GB of main memory. Without the GUI, we were able to simulate 1000 *rescue vehicle* agents, 50 *smoke* agents and 2 *helicopter* agents in approximately 5 seconds. With the GUI we were able to simulate 100 *rescue vehicle* agents, 10 *smoke* agents and 2 *helicopter* agents in 5 seconds.

5. Architecture of DMASF



Architecture of DMASF

DMASF is written in Python. This enables it to take advantage of the OOP nature of Python as well as the short development times associated with such a high level language.

The simulation follows a client server model. A server tells individual hosts (clients) what to simulate. These hosts then simulate these agents (execute user code for the agent) and then update the world state. The user can create different types of agents. Each agent type is represented by a class which the user must define and provide to the system. Therefore, an agent type can have its own set of properties as well as code for execution. In addition to this there is support for a world or environment.

5.1.The Core Classes

The *apistub* class is responsible for client side processing. It receives commands from the *apiserver* telling it what to simulate. It then allocates these tasks to *simulator* objects. These *simulators* execute the user code. Each *simulator* runs in a separate thread and is responsible for executing a set of agents.

The *apiclient* class is a daemon. It waits for connections from the *apiserver*. Once a connection is established, it downloads and deploys the simulation from the *apiserver*. It then sets up the environment and executes *apistub* which then handles the rest of the communication.

5.2.The apiserver

The *apiserver* class is responsible for handling the server side processing. It contains a *scheduler*. It also contains the handshaking code for deploying the user code on various *hosts*. It is also responsible for executing the world or environment code.

The *apiserver* main simulation algorithm

```
1. Instantiate world_handler
2. Set simtime to zero
3. Call world_handler.begin
4. while simtime less than time limit of simulation
      a. check whether simulation is paused
      b. call world_handler.beginupdate
      c. Schedule the agents
      d. Send SIMULATE to each host
      e. Wait for all hosts to finish SIMULATION
      f. Send UPDATEDB to each host
      g. Wait for all hosts to finish updating the database
      h. increment simtime
      i. call world_handler.endupdate
5. call world handler.end
6. Send QUIT command to each host
7. Disconnect all hosts
```

5.3.The Hosts and Host Manager

The host manager is responsible for each and every *host* that is connected to the *server*. Communication is done using standard BSD sockets over TCP/IP. The host manager is also responsible for sending commands to the *hosts*. It ensures reliable communication by waiting for an acknowledgement for each command.

A *host* refers to an instance of DMASF running on a computer system. Each *host* receives commands from the *server* which it executes. The commands are: (1) Simulate: this specifies the set of agents this *host* should simulate. (2) Update database: this tells the *host* to start committing changed agent states to the database. (3) Simulation Environment Changed: this tells the *host* that there has been a change such as an agent being created or killed. Each *host* has a set of *simulator* objects. These *simulator* objects run on different threads and are actually responsible for executing the user code for the agents.

A *host* can be broadly divided into two parts:

- Host Client
- Host Stub

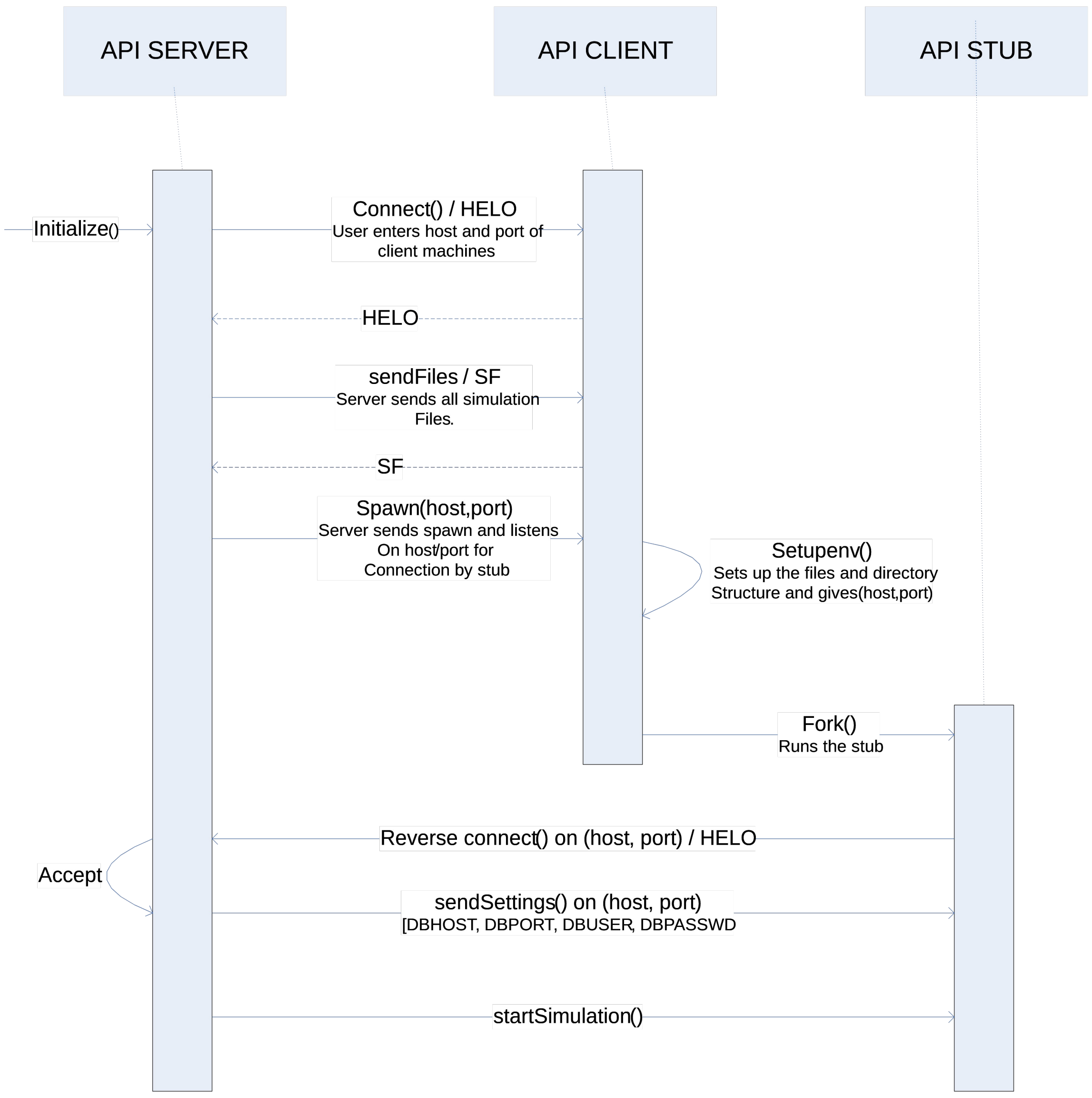
Host Client

The host client actually refers to a daemon that is running on each host. These daemons are used to deploy the simulation on that particular host. A host can have multiple daemons running on different ports. The medium of communication is TCP/IP sockets.

The initialization procedure is as follows:

- The API server first connects to the API Client and a handshaking HELO packet is sent.
- On receipt of this packet, the API Client replies back with an ACK (acknowledgement) HELO packet.
- The server then sends the simulation name.
- After this the server sends the user files (which contain the user's code).
- The client then moves these files into the deployment directory which is `/<home>/simulations/<sim name>`.
- After this the server creates a listening socket on a random port and sends the address of the server to this client.
- The client then puts this information in a settings file.
- It copies the `mas_framework` module to the simulation directory and overwrites `api.py` with `apistub.py`.
- It then starts the user program by running `python /<home>/simulations/<sim name>/main.py .`

The sequence diagram for initialization and deployment is given below.


API SERVER
API CLIENT
API STUB
Initialize()
Connect() / HELO
User enters host and port of client machines
HELO
sendFiles / SF
Server sends all simulation Files.
SF
Spawn(host,port)
Server sends spawn and listens On host/port for Connection by stub
Setupenv()
Sets up the files and directory Structure and gives(host,port)
Fork()
Runs the stub
Reverse connect() on (host, port) / HELO
Accept
sendSettings() on (host, port)
[DBHOST, DBPORT, DBUSER, DBPASSWD
startSimulation()

Host Stub

The Host Stub contains dummy methods for most api methods. The stub is required because the functions that need to be executed at the server and the client are different.

e.g. a call like `createAgent` in the user code should create an agent at the server and not at each host. Moreover, hosts require a `reverseConnect` to connect back to the server.

The sequence of steps is as follows:

- The `api` object reads the settings file which the client had written.
- Based on the settings, it connects to the server at the specified address.
- The server then sends simulation settings such as the database properties.
- The stub is now ready for user code execution.

After this, when the user code calls any functions of `api`, the corresponding stub versions are called. Of particular interest is the `Initialize` function. This is because main loop is different for both the apistub and the apiserver.

The algorithm for the apistub is as follows:

```
1. while 1:
    a. wait for Command from apiserver
    b. if cmd == CMD_SIMULATE:
        i. unpack cmd
        ii. simulate the agents
        iii. send ACK
    c. if cmd == CMD_UPDATEDB
        i. synchronize the db
        ii. send ACK
    d. if cmd == CMD_QUIT
        i. send ACK
        ii. break
2. close socket
```

5.4.The Database Layer

We use MySQL as the database. The data storage layer provides functionality required for reading and writing agent states to the database. It also provides support for storing and retrieving the messages. It is also responsible for writing agent data temporarily to secondary storage so that excessive amounts of main memory is not consumed.

Databases were used to store agent states by Kota et al [4]. The advantages of using a database are manifold. Databases are built to handle large amount of data. This is extremely useful when we need to simulate millions of agents. Using a database, looking up individual agents is extremely fast. Also, percepts can be expressed directly by giving SQL queries.

5.5.The Messaging subsystem

There is a common messaging table. Each message is stored as a tuple. A message has a *from* address representing the agent who sent the message, a *to* address representing the agent it is to be delivered to. In addition to this it contains a *simtime* representing when the message was sent and the actual message data. Messages are flushed after every iteration.

5.6.The Scheduler

The scheduler runs on the server. It is responsible for deciding what each host should simulate. The basic algorithm takes into account how long each host took to process the set of agents that it was given for the last iteration. It then attempts to come up with a dynamic scheduling policy.

The simulation can be defined as the execution of the agent code for all agents of all types. We assume that agents of the same type require approximately the same time to get processed in the same iteration on the same host for the purpose of scheduling.

This assumption is justified because agents of a similar type are expected to perform similar functions.

The scheduler has to schedule the simulation over all the hosts. The best results can be obtained when the hosts are not idle and are performing parallel computation. Thus the objective of the scheduler is to minimize the idle time (the time when a particular host is idle, while the other hosts are still computing) for all hosts.

DMASF initializes by distributing equal number of agents to each host for all types i.

Let T be the run time of a particular iteration of the simulation. Thus, T depends on the maximum run time among all the individual hosts, because the slowest host will define the run time of that iteration. The objective of the scheduler is to minimize the run time of the simulation, or to minimize T. We do this by identifying the slowest host for each iteration and then allotting it the least number of agents to simulate. Similarly, the fastest host is allotted the maximum number of agents to simulate.

The number of agents of one type allotted to one host is inversely proportional to the time it takes to simulate one particular agent of that particular type. It is inversely proportional because the faster host should be given more agents for simulation. Dividing the agents in such a ratio ensures that all the hosts finish processing their sets of agents at approximately the same time, thereby reducing the idle time.

DMASF also includes a static scheduler that schedules equal number of agents of each type on each host. The user can configure DMASF to use either of the two schedulers.

The Two Schedulers Compared

We ran the circle simulation separately using each of the schedulers. The simulation was run for one hundred thousand agents on two hosts. The first system (Machine A) had four 3GHz processors and 2GB main memory. Machine B had an AMD 3000+ (1.6GHz) processor with 512 MB main memory.

Static Scheduler

Iteration Number	Machine A		Machine B		Total Time (secs)
	Number of Agents	Time Taken (secs)	Number of Agents	Time Taken (secs)	
1	50000	19.64	50000	51.95	53.98
2	50000	19.68	50000	55.70	57.12
3	50000	19.55	50000	58.97	60.42
4	50000	19.59	50000	58.27	59.46
5	50000	19.93	50000	52.84	54.80

Dynamic Scheduler

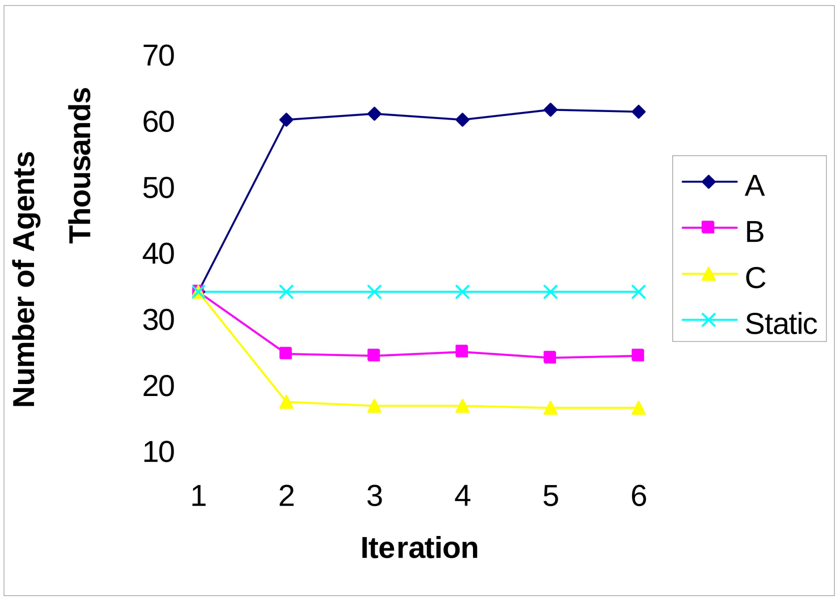
Iteration Number	Machine A		Machine B		Total Time (secs)
	Number of Agents	Time Taken (secs)	Number of Agents	Time Taken (secs)	
1	50000	19.68	50000	52.95	54.28
2	80766	31.34	19234	24.36	32.93
3	76551	30.08	23449	25.52	33.29

4	73473	28.83	26527	29.41	36.23
5	73860	28.95	26140	30.26	36.40

This shows that the dynamic scheduler gives a significant performance boost if the hosts have different computational capabilities.

The simulation was then repeated and run on three hosts. Machines A and B were the same as the ones used in the previous experiment. We added another machine, Machine C with an Intel Celeron Processor (2.6 GHz) and 512MB main memory.

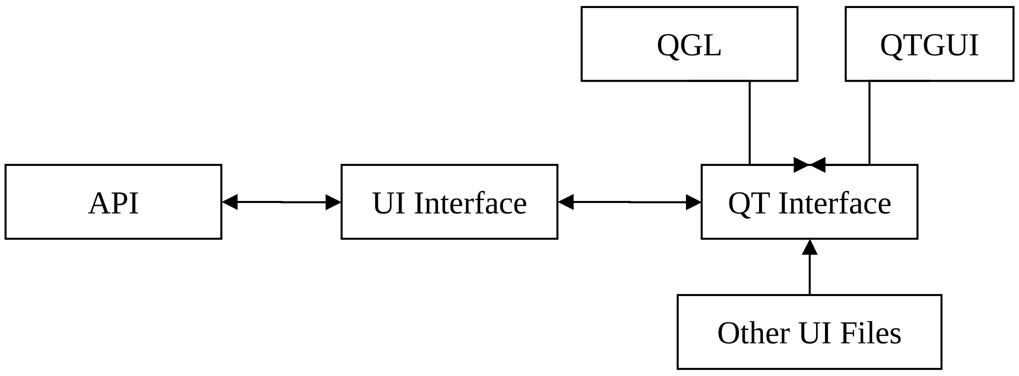
Agent allocation by schedulers



The scheduler at the second iteration allocated fewer agents for both machines B and C while allocating more agents for machine A so that the total time could be minimized. This is because the computing power of the machines was in the order of A > B > C.

5.7. The GUI

The GUI has been developed using the PyQT and Python-QtGL libraries. The GUI architecture can be described as:



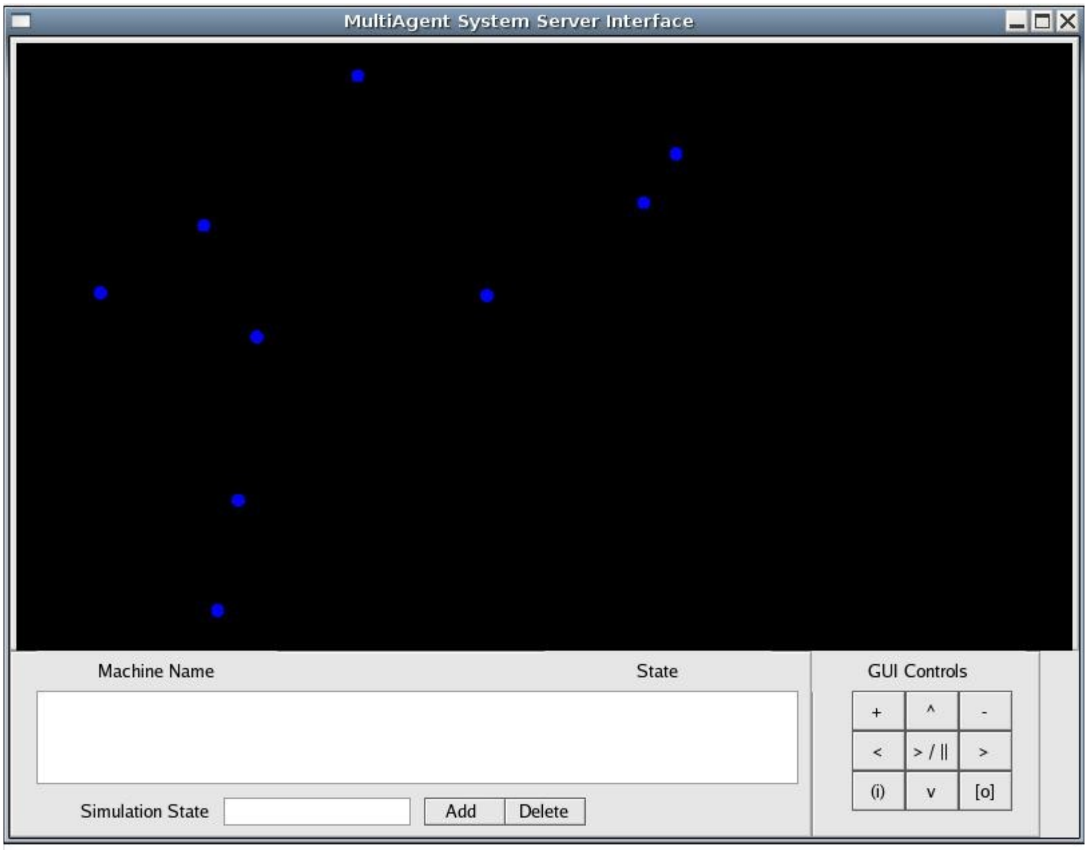
The UI Interface class has been added for proper modularity and better cohesion between the different modules of GUI and API. It acts as a GUI interface for the API and vice-versa. Some of its important functions include:

1. `changeStateOfHost`
2. `changeStateOfSim`
3. `updateGL`
4. `addHost`
5. `startSim`
6. `quit`

The QT-Interface class will define the behaviour and connections for the widgets added in the QGL and other UI classes. (UI classes are description in xml of forms generated using the QT-Designer)

The GUI runs as a different thread. An event object is used by the API to inform the UI Interface of the corresponding changes in simulation states.

Screenshots



6. Technical Challenges

During the development we faced a large number of technical challenges. Most of these challenges arose due to the sheer scale on which our simulations needed to be carried out. We feel that a list of these challenges and their solutions would be instructive to anyone attempting to build a system on a similar scale.

6.1.Optimization of the database access

One of the core aspects of this project was to simulate a large number of agents in relatively short time. Therefore, in addition to optimizing our database queries we also used advanced MySQL optimization techniques. These involved:

- Disabling and Enabling indexes on tables before agent states were updated to the database
- Utilizing the database query preparation mechanism to ensure that multiple queries did not create new connections to the database.
- Running large individual queries rather than multiple small queries
- Optimizing the agent representation to enable faster lookup.

6.2.DBBuffers

We needed a mechanism for dumping agent data to secondary storage. As agent states were updated they could not be committed to the database immediately. We could not keep this data in main memory because of the sheer volume of the data. To overcome this problem we created a *DBBuffer* class. This class would automatically dump agent data to disk whenever there was too much data in main memory. It ran in a separate thread to ensure that we could take full advantage of a multi-processor system.

6.3.Simulators

As mentioned before, each host stub has a set of simulator objects. These are actually responsible for doing the simulation. Each simulator object runs in a separate thread and handles a set of agent. This architecture enables us to take full advantage of systems with many processors. This also enables the processing of many agents in parallel, specifically when they issue blocking system calls.

6.4.Representing agent identifiers

Initially, we used a type, identifier representation for identifying agents. This technique was effective for a small number of agents. Also, it was close to the real world representation of agents. The type was a string and identifier was an integer. However, this was slowing down the retrieval and dispatch of messages as it involved a search on a text field in MySQL. By mapping individual types to integers and then packing this data into a 32 bit integer we were able to achieve a significant improvement in the dispatch and retrieval of messages.

6.5.Deletion of messages

In most simulations, we do not need to store messages until they are read. By flushing messages after each iteration we were able to optimize the message passing mechanisms. Of course, we added a function that would enable the user to disable this feature.

7. Tutorial

```
#import required modules
from mas_framework import *
import random
import math


class CIRCLE(defagenthandler):


  def update(self, simtime):
    rad = 100
    speed = 0.2
    assert(isinstance(self.db, DB))
    cx = self.db.getAggregateValue('circle', AGGREGATE_AVG, 'x')
    cy = self.db.getAggregateValue('circle', AGGREGATE_AVG, 'y')

     #updating the position and computing the theta at which I must
#move
    theta = math.atan2(self.fields['y'] - cy, self.fields['x'] - cx)
    theta+= 0.1

    targetx = rad * math.cos(theta) + cx
    targety = rad * math.sin(theta) + cy

    self.fields['x'] += speed * (targetx - self.fields['x'])
    self.fields['y'] += speed * (targety - self.fields['y'])
    self.writeState()

api = api.API()

api.db.clearDB()

api.registerAgentType("circle", CIRCLE, None, {})

api.registerWorldHandler(defworldhandler)

for i in xrange(10):
  ff = {}
  ff['x'] = random.randint(0, 1000)
  ff['y'] = random.randint(0, 1000)
  ff['size'] = 10
  ff['move'] = random.randint(0, 1)
  ff['shape'] = SHAPE_CIRCLE
  api.createAgent("circle", ff)

api.Initialize()
```

Sample code for the circle simulation

`import mas_framework`
this line imports all the libraries required for development.

`class CIRCLE(defagenthandler)`
this line defines our agent type which will be inherited from the *defagenthandler* class which is provided with the framework.

`def update(self, simtime)`

This is the update function which is defined in the *defagenthandler* class which the programmer must override. This is where the code for the agent logic is written.

```
cx = self.db.getAggregateValue('circle', AGGREGATE_AVG, 'x')
```

Here we are computing the average of the X co-ordinates of all the agents. *self.db* refers to the DB class which is provided by *defagenthandler*. *getAggregateValue* function is used to compute the aggregate value of the agent type circle. *AGGREGATE_AVG* specifies the aggregate function to apply. *x* specifies the property of the agent on which the function has to be applied.

```
self.fields
```

is a dictionary (provided by *defagenthandler*) which contains all the properties of the agent.

```
self.writeState()
```

function tells the API that the agent's state needs to be committed to the database.

```
api = api.API()
```

instantiating the API class and loading the framework

```
api.db.clearDB()
```

clearing the old database contents. If this is not called we can resume the simulation

```
api.registerAgentType("circle", CIRCLE, None, {})
```

specifying an agent type. We are creating a new type called a circle agent which is handled by the class CIRCLE. The fourth argument refers to a dictionary which contains additional properties which can be specified for the agent type. The third argument specifies a renderfunction which the system will use to draw the agent

```
api.registerWorldHandler(defworldhandler)
```

We are using the default world handler because this scenario does not require a user defined world handler.

```
api.createAgent("circle", ff)
```

We are creating an agent of the type circle with properties as defined in the dictionary *ff*.

8. Empirical Validation

8.1.Experimental Setup

The experiments were conducted on a machine with four 3GHz processors and 2GB of main memory. The MySQL version used was 5.0.18. The Python version used was Python-2.3.

8.2.Sample Worlds and Agents

8.2.1.K-P Simulation

One of the simulations created was the K-P benchmark, where K refers to the number of agents in a group. P is the number of messages sent by each agent on receiving P messages from the other agents in the group. This simulation was designed to test the efficiency of messaging.

8.2.2.Crawler Simulation

We used the “crawler” simulation as another benchmark. It had two types of agents: worker and scheduler. The scheduler assigns tasks to the worker agents by sending the worker agent a message. These worker agents then perform this task (modeled by waiting for a certain amount of time) and send the results back to the scheduler agent. We used two types of models. One model had worker agents using the system call sleep (to simulate a task such as waiting on a socket or writing to files) and the other had a busy wait. Both the models used a wait time of 10ms per agent. This simulation was designed to test the way agents were executed on each host. If an agent is waiting on a system call, the entire simulation should not wait; another agent should be simulated.

8.2.3.Circle Simulation

The circle simulation environment is where the agents are spawned in a random manner in a 2D plane. This is an extension of one of the examples provided with NetLogo [5]. The agents can only see other agents. The agents must move around in a circle whose center depends on the location of the other agents. This simulation was run on two different setups. The first setup involved two hosts running on a machine with four 3GHz processors and 2GB of main memory. We gradually increased the number of agents to see how well the simulation would scale. The simulation scaled linearly. In both the setups the MySQL database was on this machine.

The second setup involved running the simulation with one million agents. We increased the number of hosts and plotted the resultant gain in performance. The machines that were used for simulation were standard desktop computers with AMD 1.6GHz processors and 512 MB of main memory.

8.2.4.RTT Simulations

We measured the RTT (round trip time) of each message sent between a sender and a receiver agent [2]. This involved putting a timestamp on the message when it was sent and then subtracting this from the current time when it was echoed back by the receiver. For each pair of sender and receiver, we calculated the RTT from an average of 50 messages. We also measured what happens when one particular agent is flooded with a large number of messages. For this we created one receiver agent and many sender agents and observed the values of the RTT.

8.3.Results

8.3.1.K-P Simulation

The simulation was run with 10000 agents and two hosts. All times are in seconds.

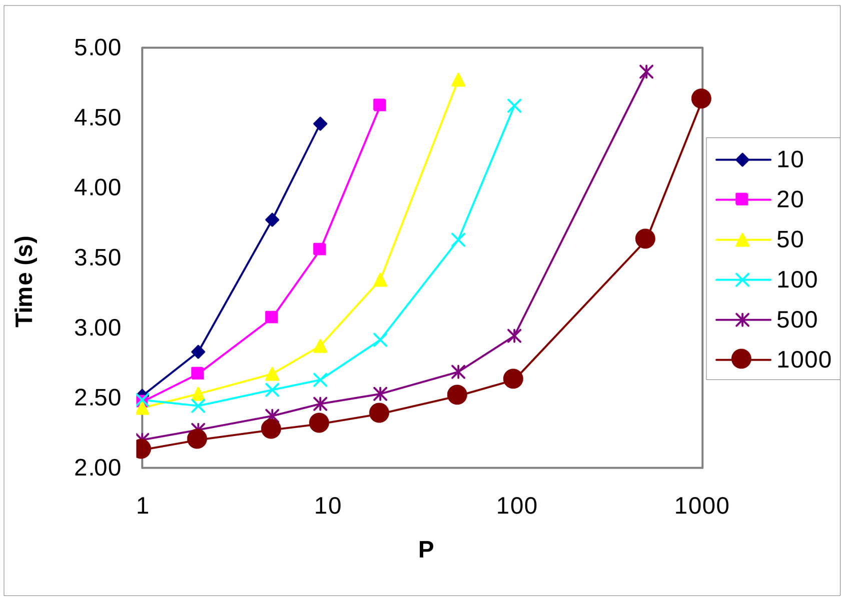
K/P Simulations for small values of K

↓K/P→	1	2	5	9	19	49
10	2.52	2.83	3.77	4.45		
20	2.47	2.67	3.07	3.56	4.58	
50	2.43	2.53	2.67	2.87	3.35	4.77

K/P Simulations for large values of K

↓K/P→	1	19	49	99	499	999
100	2.48	2.91	3.64	4.58		
500	2.20	2.53	2.69	2.94	4.83	
1000	2.13	2.39	2.51	2.63	3.63	4.63

The following graph was obtained.

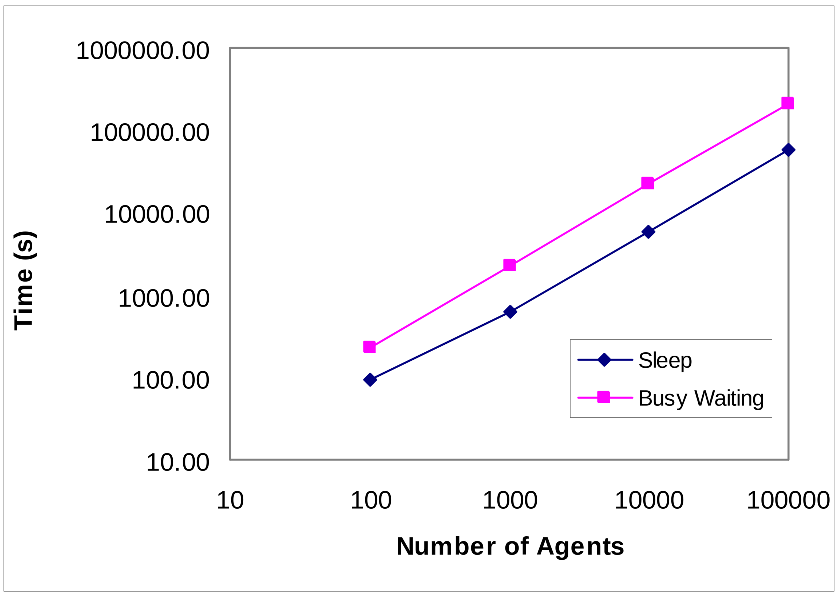


K/P Simulation

Since messages are being stored as tuples in a table, we expected a linear increase in the time with a linear increase in the number of messages in the system. It can be observed that as P increased, the number of messages in an agent group increases. Therefore with linear increase in the number of messages the simulation time also increased linearly. For the same P, with a larger K (group size) there are fewer total messages in the system and hence the time taken is less.

8.3.2.Crawler Simulation

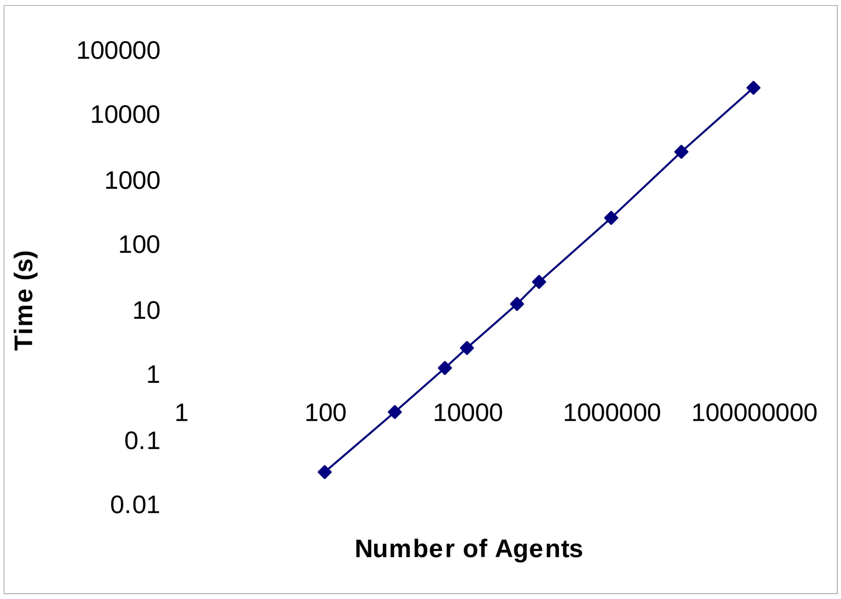
We expected a linear increase in the total time taken for each iteration. We were able to achieve this linear increase. We expected the simulation to take less time with the sleep system call as we have a number of simulators executing the agent code in separate threads. Thus, if an agent executes a system call in a simulator, another simulator would get scheduled to run. The time difference between busy waiting and using the sleep system call is clearly visible with the former taking more than two times the latter. In the busy waiting case each agent actually waited for 10 milliseconds.



Effect of Sleep and Busy Waiting on Simulation Time

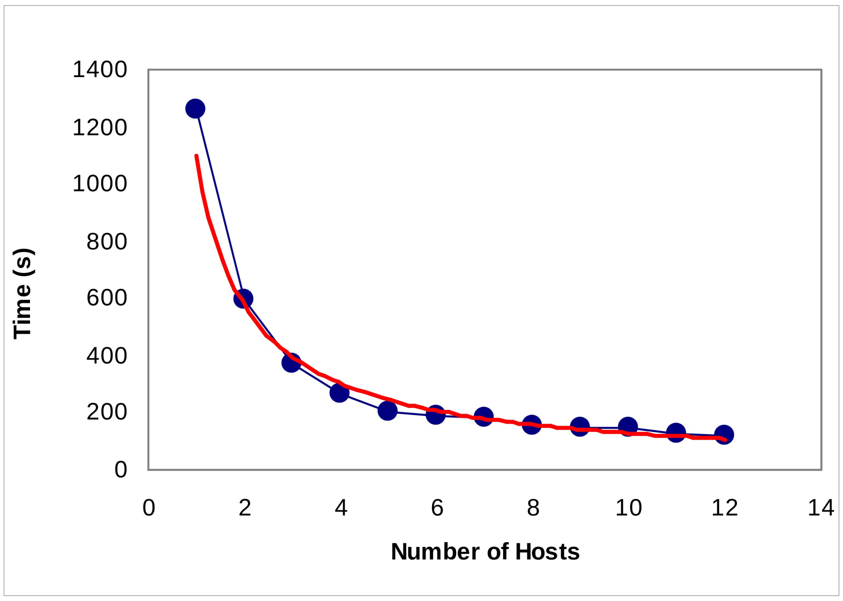
8.3.3.Circle Simulation

We got a linear increase in the time taken per iteration as we increased the number of agents. We varied the number of agents from 100 to 100 million. By extrapolating the linear results that can be seen, we can simulate one billion agents in approximately 250,000 seconds (70 hours).



Number of Agents v/s Time

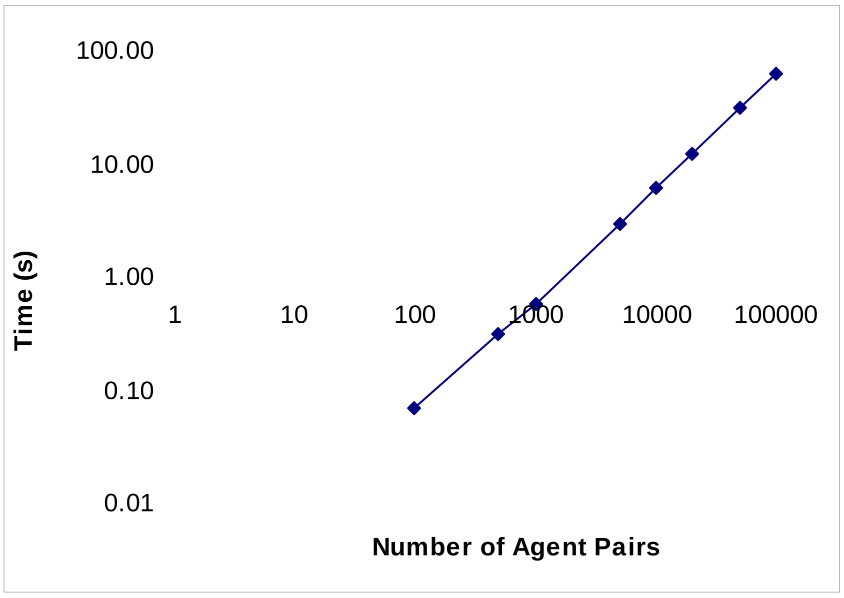
It can be noticed that the performance gain on going from one to two hosts is much more than the performance gain on going from ten to eleven hosts. With one host, we are simulating 1,000,000 agents on a single machine. With two hosts this becomes 500,000 on two machines providing a significant improvement. However, when we have 10 hosts, we are processing approximately 100,000 agents on each host. When we increase this to 11 hosts, we are processing approximately 90910 agents on a host. Thus, the performance gain in the latter case is significantly less than the performance gain in the former. In this experiment, since all the hosts had similar computational power, hence agents were allocated using the static scheduler.



Number of Hosts vs. Time

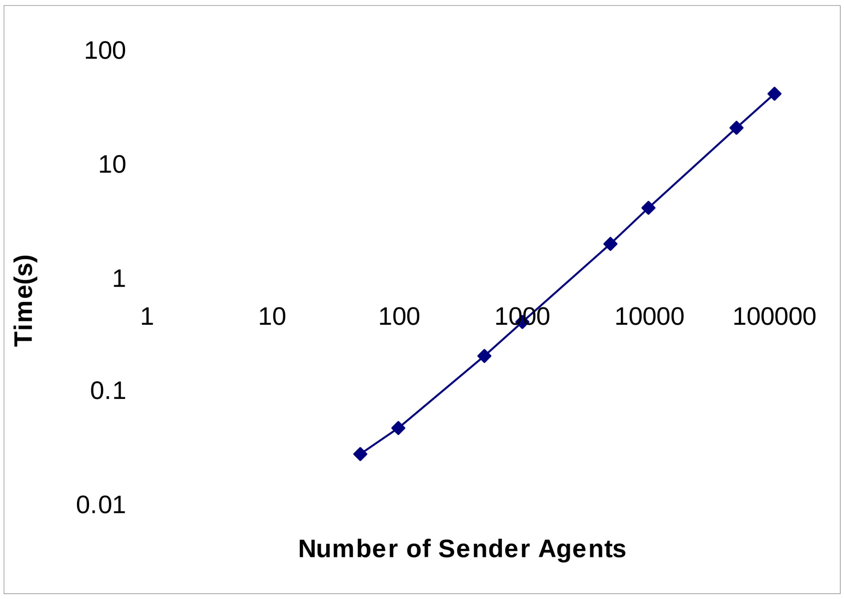
8.3.4.RTT Simulation

As expected, we got a linear increase in the RTT values. This is because as the number of agents increases, the intervals at which they were scheduled to run also increases.



RTT Values for agent pairs

We further got a linear increase in the RTT values when one agent was flooded with messages. As we increased the number of sender agents, the simulation time also increased linearly. This implies that the DMASF messaging subsystem was able to handle both the case of a large number of messages in the system as well as the case of flooding (i.e. a lot of messages being sent to a single agent).



RTT values for multiple senders and one receiver

9. Related Work

A lot of work has been done in the field of developing Multi Agent System Simulators. SPADES [1], MASON [3], NetLogo [5] and JADE [7] are some of the more popular simulation toolkits. JADE (Java Agent Development Environment) which is a middleware for developing and deploying Multi Agent System Applications. SPADES is also a Distributed Multi Agent Simulation Environment which is not language specific and allows agents to be written in any programming language. The agent code interacts with SPADES over UNIX pipes. MASON is a light, fast, scalable discrete-event Multi Agent simulation library written in Java. It also has a separate visualization layer for viewing simulations. NetLogo is a desktop simulation toolkit that scales well for small number of agents. It uses its own interpreted language for writing agent simulations. FIPA [6] is the standard for Multi Agent System Simulators. DMASF is partially compliant with FIPA standards. We are in the process of making DMASF fully FIPA compliant.

10. Conclusion

Multi Agent System technology can be used to simulate complex environments at both microscopic and macroscopic levels. Therefore, it is required to have a simulation toolkit to cater to both these needs. Many of the current Multi Agent Simulation toolkits either cater to microscopic simulation for very small environments (~10,000 agents) or do only macroscopic simulations.

One of the challenges taken up in this paper is to provide a generic toolkit that can simulate a very large number of agents in a relatively short time by using distributed computing. Our results show that we can potentially simulate one billion agents in around 250,000 seconds. With increased number of systems used for simulation it is feasible to bring down this time further.

One major advantage of our simulation toolkit is that it can be used to rapidly implement and deploy small Multi Agent System driven simulations. Thus, it is amenable for use in Multi Agent System course projects.

Since DMASF is based on Python it is easy to plug-in existing MAS decision modeling systems such as MDP into DMASF.

One of the limitations of our system is the computational mismatch between the GUI to show the simulation results and the backend that actually does the simulation. With advances in GUI rendering and related technologies it would be feasible to have real-time observation or animation of a very complex Multi Agent Simulation with one billion agents. If such GUI technology is not available then other appropriate solutions need to be found for handling this mismatch. We are currently working on simulation of very complex environments using this toolkit.

11. References

Reference Source	Details
[1]Research Paper	• Patrick F. Riley, George F. Riley • *SPADES – A Distributed Agent Simulation Environment with Software-in-the-Loop Execution* • Proceedings of Winter Simulation Conference, 2003
[2]Research Paper	• Luis Mulet, Jose M. Such, Juan M. Alberola • *Performance Evaluation of Open-Source Multiagent Platforms* • International Conference on Autonomous Agents, 2006
[3]Research Paper	• Sean Luke, Claudio Cioffi Revilla, Liviu Panait, Keith Sullivan • *MASON – A New Multi-Agent Simulation Toolkit* • Society for Computer Simulation International, 2005
[4]Research Paper	• Ramachandra Kota, Vidit Bansal, and Kamalakar Karlapalem • System Issues in Crowd Simulation using Massively Multi-Agent Systems. • Workshop on Massively Multi Agent Systems, 2006.
[5]Research Paper	• Seth Tisue, and Uri Wilensky. • Netlogo: Design and Implementation of a Multi-Agent Modelling Environment. • Agent2004 Conference.
[6]Internet Site	*The Foundation for Intelligent Physical Agents(FIPA),* http://www.fipa.org [url valid as on 27th October, 2006]
[7]Research Paper	• Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa • *JADE – A FIPA compliant agent Framework.* • Proceedings of The Practical Applications of Intelligent Agents and Multi-Agent Technologies (PAAM), 1999.
[8]Internet Site	Python Programming Language Documentation http://www.python.org/doc/ [url valid as on 27th Oct, 2006]
[9]Research Paper	• Gaku Yamamoto • *Agent Server Technology for Managing Millions of Agents* • Proceedings of Massively Multi Agent Systems 2004
[10]Book	• Abraham Silberschatz, Peter Galvin, and Greg Gagne • Operating Systems Concepts, Sixth Edition • Published by: John Wiley & Sons, Inc. 2001.
[11]Book	• Stuart Russell, and Peter Norvig. • Artificial Intelligence: A Modern Approach. • Prentice Hall Series in Artificial Intelligence, 1995.