Study of Mobile Agent Algorithmic Implementation and Development

Scotty Smith

September 15, 2006

1 Introduction

A mobile agent is a piece of software that can move itself from computer to computer over a mobile agent enabled network. These agents are completely autonomous, performing their set tasks and then reporting the results to the end user. For this system to work, all of the computers that the agents are going to be executed on must be running a mobile agent protocol.

Aglets is a Java based mobile agent protocol. It allows for the creation of mobile agents, which can then perform various tasks. The development of Aglets is an on-going process, since Aglets is an open source program. It is complete enough to start developing applications that can use these mobile agent protocols.

Along with the development of the mobile agent protocols, various algorithmic issues have been identified. These issues have been studied, and provably correct algorithms have been developed to alleviate the issues. No where in the literature, however, has anyone stated that the newly developed algorithms have been implemented into a real mobile agent protocol.

The purpose of this project was to implement one of the more prominent algorithms that has been developed, the black hole search algorithm, into a mobile agent protocol, Aglets. The purpose of this was to see how easily the new algorithms could be implemented into real-world scenarios, as well as testing the algorithmic claims of the algorithm creators.

In Section 2, we will discuss the hardware setup we used for this research. In Section 3 we will discuss the install of the Aglets protocol. In Section 4 we will discuss the development of mobile agents using the Aglets protocol. In Section 5 we will briefly discuss the black hole search algorithm for ring networks. In 6, we will briefly discuss the black hole search algorithm in

more general networks. In Section 7, we will discuss the implementation of the black hole search algorithms in the Aglets protocol.

2 Installing the Cluster

For this project, it was decided that a cluster would be the most beneficial place to install the Aglets protocol. We had previously discovered a version of Linux called Rocks, which focused on creating a cluster. We decided to install this on our hardware cluster.

The install on the master node was pretty straightforward. The real task was getting the slave nodes installed. These nodes only have internal hard drives. They do not have any optical storage devices. Luckily, Rocks supports PXE booting to install the nodes.

When the first node completed installing, a problem was noticed. These nodes also do not have any video output, and the nodes were hanging up well before they booted into the operating system. We attempted to watch the boot from minicom, but when the boot loader, grub, tried to load, we lost output and the node restarted. Grub seemed to be the problem. To fix this problem, we needed to have someplace for the output from grub to be dumped. To achieve this, we added the line "console=ttyS0, 115200n8" to the grub.conf file on our master node. This caused grub output to be outputted to minicom. since grub.conf on the master node is where the slave nodes get their grub information, so this change solved that problem, after reinstalling Rocks on the slave node.

Now that we could see the output through minicom, we continued to install nodes. We soon encountered another problem, however. The nodes would stay up for a short amount of time, but soon they would shut down. There was a message on the nodes stating that INIT was reloading. To solve this, had to make more changes to the grub.conf file. This time, we added "no apci apic off". After more reinstalls, this problem was solved as well.

The final problem we had to overcome was several hardware issues. Most notably, we needed to have enough nodes up to constitute a large enough network to run our test cases on. We had several nodes from the cluster that had severe hardware issues. We were lucky enough to get 6 of the actual cluster nodes hooked up and installed, and we had a spare computer that we were able to install the Rocks software on, so we ended up with 7 nodes to work with.

While 7 nodes were adequate for our smaller test cases, we needed more computers for our larger test cases. We have access to a lab of 24 computers,

which we decided would be adequate for our test cases. These computers did not get rocks installed, as they were already running Ubuntu Linux.

3 Aglets Install

The install of the Aglets software was very straightforward. After downloading the install package from *http://aglets.sourceforge.net/*, we extracted the jar file. From there, we followed the included instructions to get Aglets installed on the master node. This was very simple, but it still took some time. For the install on the slave nodes, we decided to write a shell-script that would do the install for us.

The shell-script was written following the same steps that were done to install on the master node. As a result of this, Aglets was up and running on all of the nodes in no time. The shell script turned out to be very useful. Any time one of the slave nodes of the cluster is improperly shut down, rocks is reinstalled. The hard drive is reformatted. With the script, reinstalling was not a hassle.

There were intermittent problems with the install, however. The first attempt to install Aglets on the master failed due to a missing file. Simply re-downloading the package solved the problem, but occasionally similar files are "missing" from this package, which are simply solved by re-doing the install process.

To get Aglets installed on the lab computers, we took a different approach. The lab computers have access to shared directories on an external server. By installing Aglets in one of the shared directories, we were able to run Aglets on all of the lab computers with just one install.

The only other problem that we faced was the lack of a graphical interface for the Aglets protocol to output to. The Aglets run script did not have a help file, so after reading through the FAQs carefully, we discovered that there were a few useful command line arguments that could be used, such as -daemon, which starts a server that does not accept user input. The one that we most commonly used, however, was -nogui, which caused all output to go to the console.

4 Development of Mobile Agents in Aglets (see [3])

Developing mobile agents in Aglets is a fairly straight forward process. Each agent has to extend *Aglet*. From there, you have to override a few basic

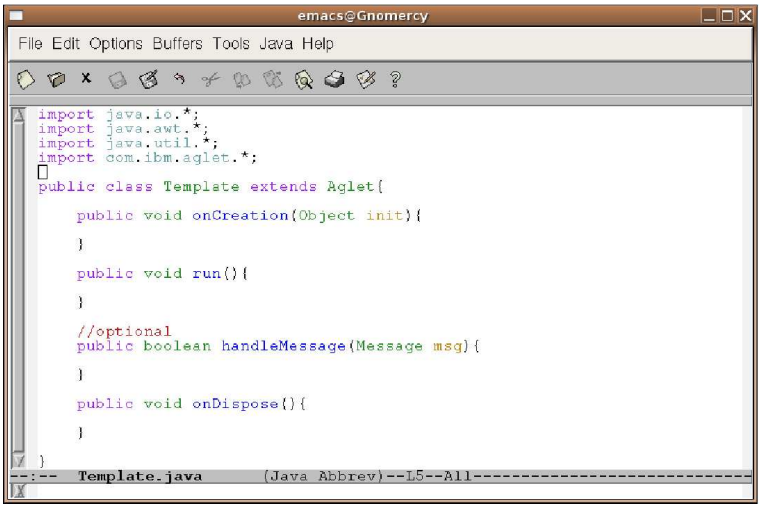

Figure 1: Typical Aglets file template

functions. The first of which is `onCreation(Object init)`. This function is called, as its name implies, when the agent is created. It is called once, and is never called again. This is similar to a constructor, so you can do similar things that you would in a constructor. The next method is `run()`. Run is called every time the agent arrives on the server. This is usually where a bulk of the computations occur when developing software using the Aglets protocol. Also, you may wish to override `onDisposing()`. This is called when the agent is being disposed of.

Another method you may wish to over-ride is `handleMessage()`. This returns a boolean value indicating whether or not the message was handled by the agent. If false is returned, an exception is thrown stating the message was not handled. While this method is not necessary in some cases, it can be important in others. Some of the provided tools, such as the *SeqPlanItinerary* class, use agent communication messages to direct an agent to perform a certain task. Depending on what is being implemented, this function can possibly be ignored.

To have an agent get transferred from server to server, you have to use what is called the *atp* (aglet transfer protocol) address of the server. This is usually just the host name of the computer running the server, but in the cases that it is not, the atp address is easily obtained from the running server itself. Alternatively, you can use the IP address of the server. The agents expect the address in the form *atp://hostname:port/*. If the default

```
public boolean handleMessage(Message msg){
    if(msg.sameKind("travel")){
        try{
            System.err.println("Recieved Travel Message");
            //update the updatingIndex to the the index of this server.
            AgletProxy proxy = this.getProxy();
            updatingIndex = dns.getIndex(proxy.getAddress());
            System.err.println("updatingIndex = " + updatingIndex);
            System.err.println("Adding GOING message to whiteboard: " +
                               itinerary.getCurrentAddress());
            wh.addMessage("Master Going " + dns.getIndex
                          (itinerary.getCurrentAddress()));
            return true;
        }
        catch(Exception e){
            System.err.println("EXCEPTION! " + e);
            e.printStackTrace();
            return true;
        }
    }
    else if(msg.sameKind("update")){
        //at this point, we need to update the whiteboard
        System.err.println("Master: Should be updating the whiteboard");
        //add message "Safe _" to the whiteboard
        wh.addMessage("SAFE " + updatingIndex);
        System.err.println("Added 'SAFE " + updatingIndex + "' to WB");
```

Figure 2: Example of the handleMessage method

port is being used, the port number is optional.

5 The Black Hole Search Algorithm for Ring Networks

The *black hole problem* is one which directly impacts the capabilities of the mobile agents. A black hole is a node in a network which destroys, without a trace, anything that enters it. This is a very big concern, since any data collected by the agent is destroyed upon entering the black hole. Therefore, if you know that there is such a black hole in a network, it is very important to be able to identify exactly which node is the black hole.

Dr. Nicola Santoro of Carleton University is one of the leading researchers in algorithm development for mobile agents, and the black hole problem is one which he and his co-authors has solved. As solutions, he devised different strategies varying on the structure of the network. In [1], the black hole problem was studied on a ring network. In this network topology, [1] shows that two agents are necessary and sufficient to locate the black hole, assuming that there was a whiteboard on each server where agents can leave messages for other agents. To accomplish this, the ring is divided in half. Consider the situation is figure 1. Node 0 is the home base of the agents. Agent 1 would follow the path $0, 1, 2, \ldots$, and agent 2 would follow path $n, n-1, n-2, \ldots$. When an agent successfully enters a node, it returns back to the previous node to record its success by leaving a "safe" message on the whiteboard. This is referred to as the cautious walk. Eventually, one of the agents will enter the black hole. Assume without loss of generality that agent 1 finishes its half first, and agent 2 is stuck in a black hole at node $n-x$. Agent 1 then travels over the safe path (the path which contains

only nodes that have been labeled "safe") in order to locate agent 2. Once agent 1 arrives at a node that does not have a safe message on it, the last node safely explored by agent 2, agent 1 will know that agent 2 was trying to explore node $n - x$. Agent 1 divides the remaining nodes ($n/2$ to $n - x$) in half, and writes on the whiteboard of the current server which nodes it will be taking over, and travels to explore these new nodes by traveling back over the nodes which are definitely safe. This process continues until there is one node left unexplored, which is the black hole.

6 Black Hole Search Algorithm for Arbitrary Networks

While [2] advertises arbitrary networks, there is still a constraint as to which networks the black hole search can actually be solved. The network graph must be two-connected. This is important, because with this constraint, there is always a clear path around the black hole in the network.

In [2], several different algorithms are provided for locating a black hole in such a network. These algorithms differ in the amount of knowledge that the agents have about the network. The first algorithm they provide assumes total ignorance of the network (agents only know the number of nodes in the network), while the last algorithm assumes complete knowledge of the network (agents have a complete "map" of the network).

The complete knowledge algorithm is fairly similar to the black hole search in the ring network, even requiring only 2 agents. To start, the network is divided into two disjoint, but connected, subgraphs. Each agent takes a different subgraph, and performs a cautious walk on its subgraph. One of the agents is going to enter the black hole.

The existence of such a partitioning scheme is proven in [2]. The partitioning algorithm provides a way to separate the "unsafe" portion into disjoint connected subgraphs, such that each subgraph contains an edge to the safe portion of the graph. This way, we know that we can get to each of the subgraphs, and we know that all the nodes can possibly be reached.

When an agent completes its portion of the network, it tries to locate the other agent. Once it finds the last safe node that the other agent successfully explored, the remaining portion of the network is partitioned into subgraphs. This continues until only 1 node is left unexplored, which is the black hole.

```
import java.util.*;
import java.awt.*;
import java.io.*;

public class Whiteboard{
    //Vector of messages
    private static Vector messages;

    public static Whiteboard wb = new Whiteboard();

    //constructor for whiteboard initilizes the messages vector
    public Whiteboard(){
        messages = new Vector();
    }

    //get instance returns the static whiteboard
    public static synchronized Whiteboard getInstance(){
        return (wb);
    }
```

Figure 3: Excerpt from the Whiteboard class

7 Implementation of the Black Hole Search in Aglets

7.1 Ring Network Algorithm

The first task we decided to tackle was how to implement a black hole. Our first idea was to create an agent on a server that would continually broadcast a message. When an agent on this node receives this "black hole" message, it would stop traversing the network. We decided to try this, thinking that it would be easier to keep track of the agents if they did not get destroyed. We had success with simple test cases with this model, but as stated later we had to make some modifications.

It is important to note that there are some interesting things that happen when working with messages in Aglets. The first thing to note is that, when simply using the `sendMessage()` method, the agent will freeze until it receives a reply back from the recipient agent. To do this, the recipient must call the `msg.sendReply()` method, which can also take several parameters. It is possible to get around having to do this by sending asynchronous messages. To do this, the `sendOnewayMessage()` method must be called instead.

The next problem was that we needed two agents, but by the time that we could create the second agent, the first agent might be done. To alleviate this, we had the first agent create the other agent. This was a fairly easy process, since the Aglets API directly allows for this.

Another problem we faced was the fact that Aglets allows for only direct communication between agents. What we needed for this algorithm, however, was a whiteboard, where agents could leave messages for other agents. It was simple enough to write a whiteboard class. We just created an object

that was simply a vector of messages, with a simple `addMessage()` function. The hard part was getting this to be a standard part of the server. We first had to download the source code for Aglets. The whiteboard class files were included in the same directory as the Tahiti server (the default Aglets server) java files, and the creation of the whiteboard was added to the Tahiti server (see 4). The new server was compiled, and replaced the old class file in the jar file. The whiteboard class was also added into the jar file.

Along with the whiteboard, we needed the agents to know the atp addresses of the servers. To do this, a DNS class was written to be included in the servers just like the whiteboard. The DNS is very simplistic, associating integers with the atp addresses of servers. The servers get this information from a local file. The DNS allows for the algorithms to become less complex, since then only integers have to be dealt with, instead of the whole address.

Note that these additions are for the servers themselves, and not the agents. As such, there was no need to make these classes serializable. Any class that holds information that the agent needs as it travels from server to server has to be serializable.

Next we had to come up with some way of developing a path for the agents to follow, and let them know when to update the whiteboards. The *SeqPlanItinerary* class worked very well for this. The *SeqPlanItinerary* class lets you set up an itinerary for the agent, and have a specific message broadcast to the agent when it arrives on the specified server. This allowed us to use the `handleMessage` method to differentiate between traveling to a node, and traveling back to the home node, a time in which the whiteboards need to be updated.

The *SeqPlanItinerary* class inherits many methods from the *SeqItinerary* class. One such method is the `goToNext()` method, which will send the agent to the next destination in the itinerary. There does not need to be an explicit call to this function in order for the itinerary to complete. Once the

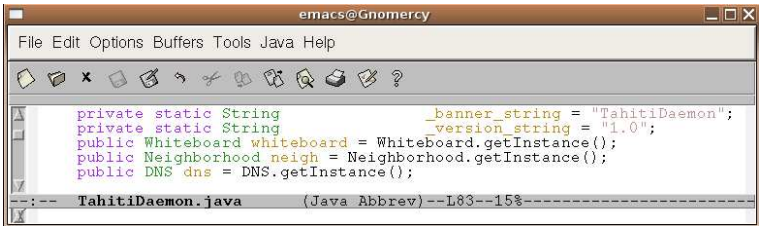

Figure 4: How additions were added to the Tahiti Daemon

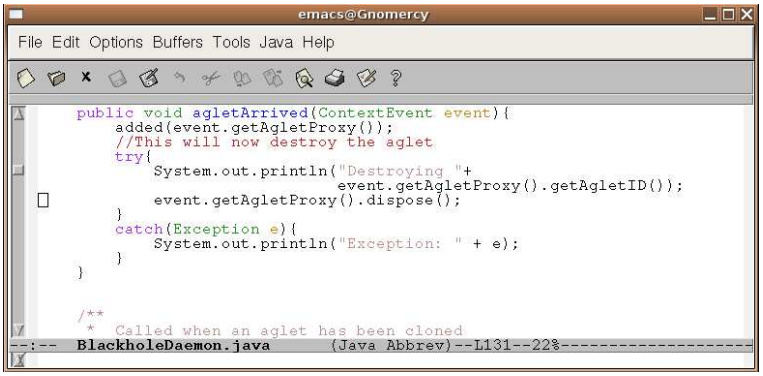

Figure 5: Changes to create the Blackhole Daemon

`startTrip()` function is called, the agent travels to each node in its itinerary, and upon arriving at a node, executes its run and handleMessage methods. Once the execution of this is complete, the agent will automatically travel to the next destination. Calling the `goToNext()` method will most likely result in exceptions being thrown.

With the initial paths set up, we decided to do some simple test runs. In the first run, we just wanted to make sure the paths were working. They ended up working as expected. The next trial was one with a Black Hole. Unfortunately, the agents traveled so fast that the "black hole" message was not getting processed. This means that we had to come up with something different for the black hole.

We tried to stick with having an agent on the server do the work for us. We developed an agent to try to destroy the incoming agents, but the agents were still traveling too fast. After exploring the API, we found a context listener, that would detect when something arrived on a server. Unfortunately, the context listener can only be implemented by a server. We had to once again reprogram a server (see 5). This time, instead of directly altering the files, we created new files so that that we could have both the normal and black hole servers accessible at the same time by having two different startup scripts. While we were able to create the black hole server, and it works as one would expect a black hole server to work, we could only get it to operate as a daemon server. This causes no problem with the algorithm, however.

With our black hole server in place, we had to tackle the next phase of the algorithm. At this point all that was left was, once an agent completes its half of the network, to locate the other agent, and recompute itineraries.

This ended up being pretty easy, based primarily on the practice that we had with programming the agents. Checking the whiteboard for safe messages was an easy task. We had some slight errors when it came to timing when messages were going to be handled, which caused some exceptions to be thrown when trying to locate the other agent, but these did not affect the algorithm's performance.

In [1], the run time of the ring algorithm is claimed to be $O(nlogn)$, based strictly off of the travel time of the agent. The run time of the calculations the agent performs on each server was assumed to be $O(1)$. In our implementation of the algorithm, our travel time of the agents matches that of the claim, $O(nlogn)$. However, Our findings show that the calculations performed by the agent are actually $O(n)$, which causes the actual runtime of the algorithm to increase.

7.2 Arbitrary Networks

Once we had successful runs with our ring network algorithm, we then decided to try the arbitrary network algorithm, assuming complete knowledge of the network. This saves a bit of programming, since we would not have to implement anything extra over what we already had.

It was at this point that we started getting weird errors with our DNS class. The DNS reads in its entries from a file located in the shared directory on the cluster. This allows for only having to change the DNS setup in one location, rather than on each individual node. We started running into permission issues when we started the general algorithm. After some searching, we discovered that it was not the system permissions, but the Aglets policy permissions for file access. A quick change to the *.aglets/security/aglets.policy* file, done on every server, gave us the correct permissions.

Up to this point, we did not have, or need, a remove message function for the whiteboard. Due to all of the updates occurring on the whiteboard, we actually need this functionality. This was fairly easy to implement, since our whiteboard class was simply a vector of messages.

One of the last issues facing us was how to partition the network. The agents have complete knowledge of the network, so from the home node, we could compute the partitions. Our partitioning algorithm is a very inefficient algorithm. It simply computes a potential partition, and runs it through a series of checks to validate the partition. It does successfully compute the paths however. After implementing our partition algorithm, we found the sources for the proven partitioning algorithm, which is considerably more

efficient. Luckily, our partition algorithm ended up being general enough to use at each place where a re-partitioning was required.

We also needed to have some way of computing the shortest paths between two nodes. We decided to program *Dijkstra's algorithm* in order to accomplish this. A slight modification of the input to the algorithm allowed us to compute the shortest safe path between the nodes.

Once all of the programming issues were solved, the algorithm was fairly easy to program. There was a lot of coding to be done, but it was all fairly simple.

8 On-Going Research

One thing that we hoped to accomplish with this project was to come up with a general solution for finding n black holes. Our hypothesized algorithm is merely a slight modification to the algorithm proposed in [2]. The network will, however, need to be $n+1$ connected, since there has to be a safe path around the black hole to guarantee a successful find. Here is what changes we believe will work for a graph that is $n+1$ connected:

I. n+1 agents are required.

II. The partitioning algorithm creates n+1 partitions.

III. For the cautious walk, the agents must always travel back to the home node to leave a safe message. This is because now we simply cannot locate the other agents, since there are so many of them.

IV. Repartitioning occurs at the home node also, which acts as a central location for all of the agents during execution.

As far as partitioning is concerned, we believe that one exists. Our goals for the future are to 1) show the partitioning exists, and 2) prove our algorithm works on all $n+1$ connected graphs.

9 Conclusions

The installing of the Aglets software was very straightforward. Other than the few intermittent problems we encountered, we were able to get Aglets installed on all of the servers in one day. The time it took to implement the Black Hole algorithm took much longer, as was expected.

The Black Hole algorithm is fairly simple, but it relies on some features that the Aglets protocol did not have. The most notable of which is the whiteboard. While in this case, it was very simple to create and implement a whiteboard into the protocol, this may not be the case with others. It would also be simple to create an agent on each server to act as a whiteboard, but in a real world situation, you would either have to have the agent always up, even when not performing the black hole search, or you would have to create the whiteboards as you successfully entered a server.

Another feature that the Aglets API did not provide that would have been useful for our algorithms would be a way to detect other atp servers that are in communication with the current server. While this would not have helped the ring network algorithm, this would have helped out a lot in the more general algorithms. In the algorithm that we chose to implement, we were able to work around not having this feature through the DNS functionality we programmed, along with the knowledge the agent has about the network. The other algorithms have considerably less knowledge about the network, hence cannot create itineraries, nor do they have a way to figure out what servers they can travel to. They required some feature such as this. We would have to program in the functionality ourselves.

One last feature that would be useful in the future would be a way to create x number of agents, and then have them start executing. We got around this in our implementation because we only ever needed two agents. It was a simple process to have the "master" agent to create the "slave" agent. In other algorithms that need x agents, the process becomes more complicated. This functionality would be very useful, especially with the on-going research.

The rest of the algorithm was fairly easy to implement. The Aglets API provides a lot of useful functions and classes. After discovering the power that Aglets provided to the programmer, it was fairly easy to finish programming the algorithm.

References

[1] Stefan Dobrev, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Mobile search for a black hole in an anonymous ring. *Lecture Notes in Computer Science*, 2180:166–??, 2001.

[2] Stefan Dobrev, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Searching for a black hole in arbitrary networks: optimal mobile agent

protocols. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 153–162, New York, NY, USA, 2002. ACM Press.

[3] Luca Ferrari. The aglets 2.0.2 user's manual, October 2004. https://www.sourceforge.net/projects/aglets/.