A Chat Interface for Human-Agent Interaction in MAST

Marco Carvalho, Matteo Rebeschini, James Horsley and Niranjan Suri
{mcarvalho, mrebeschini, jhorsley, nsuri} @ihmc.us
Institute for Human and Machine Cognition
40 South Alcaniz St., Pensacola, FL 32502 USA
850-202-4446

ABSTRACT
In this paper we introduce a group-messaging interface that allows humans to efficiently interact with a group of agents through a hierarchical and customizable text protocol. Our approach is presented in the context of the MAST mobile agent-based framework for security and administration of large scale computer networks. The MAST framework is primarily human-centric and directly supports human-agent interaction that enables customized agents to notify administrators and react to abnormal environmental conditions. The proposed IRC-like interface was developed and tested in the context of MAST. In this paper we present the group-manager interface in contrast with other agent interfaces currently available in the MAST framework.

Categories and Subject Descriptors
I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence – *multiagent systems, intelligent agents.*

General Terms
Management, Design, Reliability, Security, Human Factors.

Keywords
MAST, Network Security, mobile agents, HCI, group messaging, NOMADS, IHMC.

1. INTRODUCTION
Intelligent software agents have long been proposed for distributed complex tasks such as network and systems management. In most cases, proposed agent-based frameworks tend to rely primarily on BDI-like agents[1] designed to sense, reason, and act upon changes in the network and system environments on behalf of system administrators [1-4].

There are many AI issues related to the problem, but one important point to be made is that network and system management is often based on sequences of simple tasks that are repetitive, error-prone, and greatly based on complex human reasoning. In general, experienced system administrators know the course of action to be taken to address specific issues, but are often overwhelmed with the complexities involved in executing the tasks on large scale heterogeneous networks. A classical example of this phenomenon is observed on a regular basis, when security officers and network managers receive CERT® security advisors that are relevant to their network and simply fail to take the recommended corrective measures, usually because of the significant effort and likelihood of errors involved.

Realizing the potential of software agents as simple self-contained enforcers of commands and policies established by system administrators, researchers started to propose alternative approaches for agent-based management tools. The MAST project [5] [6], for instance, was introduced to provide a human-centric approach to the problem of network and systems security, that is, an approach that primarily relies on human-initiated actions and expertise, with less autonomy from the part of the agents. MAST is a mobile-agent based security tool for training, monitoring, and administration of security-related tasks in corporate networks. In MAST, mobile software agents are primarily used to extend and improve the human capabilities in large scale systems. The tool was designed to enable consistent and context-dependent tasks to be easily executed (and verified) in large scale computer networks. In general, MAST agents can be easily coded and modified directly by system administrators, with no specialized knowledge about agent mental models, high-level agent communication languages, and other intelligent agent concepts.

Building from an analogy to security guards roaming through a building at night to ensure that all doors and windows are locked, MAST agents are generally designed to reliably accomplish simple tasks with full coverage. Much like the security guards in our analogy, if a "door" cannot be locked, the guard (security agent) will notify its supervisor (system administrator) who will then make the appropriate decision on how to handle the "anomaly".

The framework provides numerous facilities for human-agent interaction to allow agents to easily locate and notify specific system administrators of potential problems. However, for large systems, the number of active agents requesting human assistance can still be overwhelming as users have to individually handle each special case.

In this paper, we propose a chat-like interface that allows a system administrator to interact with multiple agents simultaneously through a text protocol. The interaction protocol is extensible and can be disclosed at run-time, allowing security agents designed by one system administrator to be used, and modified, by other administrators.

[1] A BDI agent is a particular type of rational software agent, extended with some mental states: Beliefs, Desires, Intentions (BDI).

SAC'06, April, 23-27, 2006, Dijon, France.

In the next section, we will briefly describe the MAST system, followed by the proposed Group Manager interface and an illustrative scenario describing its applications. We then present and discuss some preliminary experimental results, followed by our conclusions and future work.

2. THE MAST ARCHITECTURE

MAST is designed to operate as a distributed system. It integrates and builds upon two key technologies: mobile software agents [7] [8] and concept maps [9] (for knowledge sharing). Concept maps are tools for organizing and representing knowledge. They have been widely used in the many different domains since their inception in the 70's by Novak [10]. In MAST, concept maps are used for building and sharing domain knowledge between system administrators. Mobile agents are used to move state and computations to hosts on the network in order to execute locally and perform the necessary monitoring and management tasks

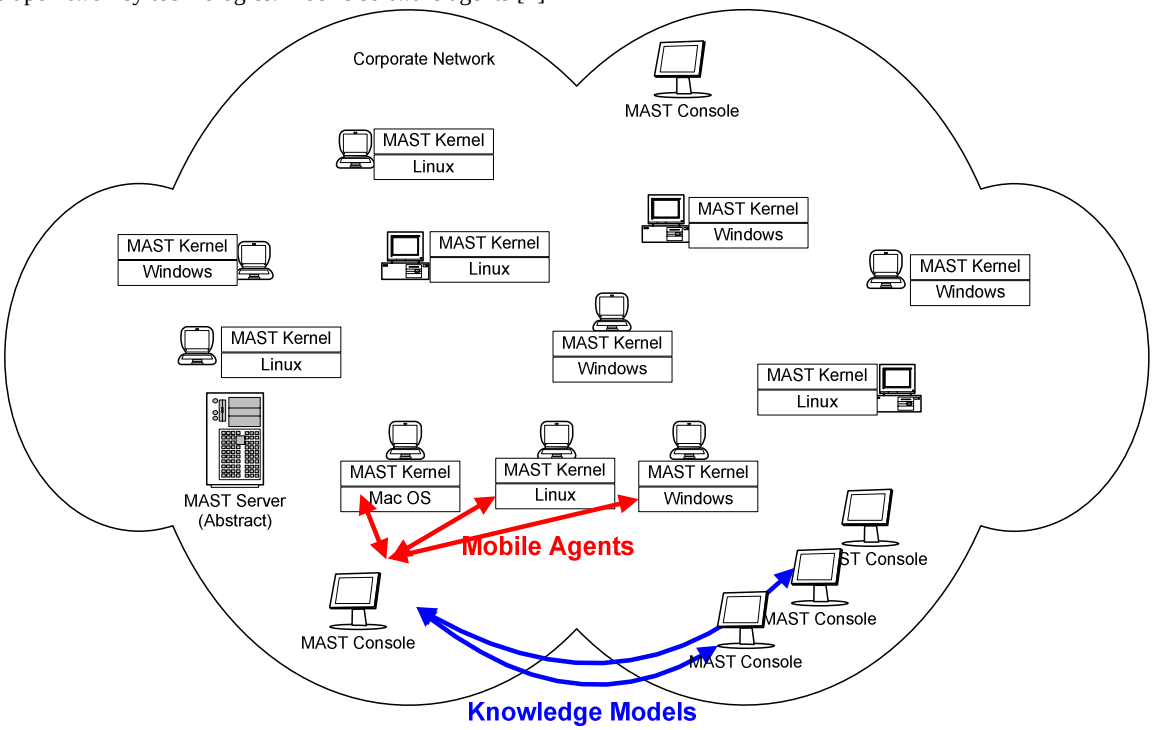

Figure 1 – Sample MAST network

Figure 1 illustrates the core components in MAST that are relevant for this work, namely, the security kernel, MAST server, and the MAST Console.

• The Security Kernel is installed on all network hosts and forms the foundation of the security substrate, handling the execution of the software agents. Furthermore, the kernel abstracts the underlying system and application-dependent details of each host. In addition to the standard cross-platform Java API, the MAST kernel also includes an extended cross-platform API that supports a number of common security-related tasks such as account, process, and file-system management.

• The MAST Server is an abstract entity. The server is composed of a distributed set of agents that might reside in more than one host but collectively react as a centralized entity.

• The MAST Console is the graphical interface that provides access to the system. Based on user credentials, the console allows different levels of access to agents and systems. The console also includes facilities for high-level agent communication and an embedded editor to modify and create new agents.

Communications between all these components is always encrypted using TLS at the infrastructure level. Every kernel, console, and server component is manually registered with the framework, using out-of-channel key exchange. This initial one-time overhead is necessary to bootstrap the system.

A snapshot of the MAST console on a small network is shown in Figure 2. The list of computers in the network (also referred to as kernels) is shown in an internal window at the upper left corner in the console. The list of kernels provides a real-time view of all hosts in the network, including their running state and the quarantine status. A kernel is automatically quarantined by the framework if there is sufficient evidence from the server components (or specialized monitoring agents) that the security kernel in that specific host can no longer be trusted. In these cases, the framework isolates quarantined kernels and immediately notifies the system administrator. From the list of kernels, the user can select the target machines to dispatch security agents.

Once dispatched, interactive agents can remotely create a graphical interface to communicate with the user. Examples of when this interface could be used are: the agent's operation requires user input, local conditions at the host (e.g. a file lock) prevent the agent from carrying out its task, and sending report data to the user. Figure 2, for instance, shows and example of interactive agents where each agent launched to the highlighted kernels reports back a list of local processes for termination.

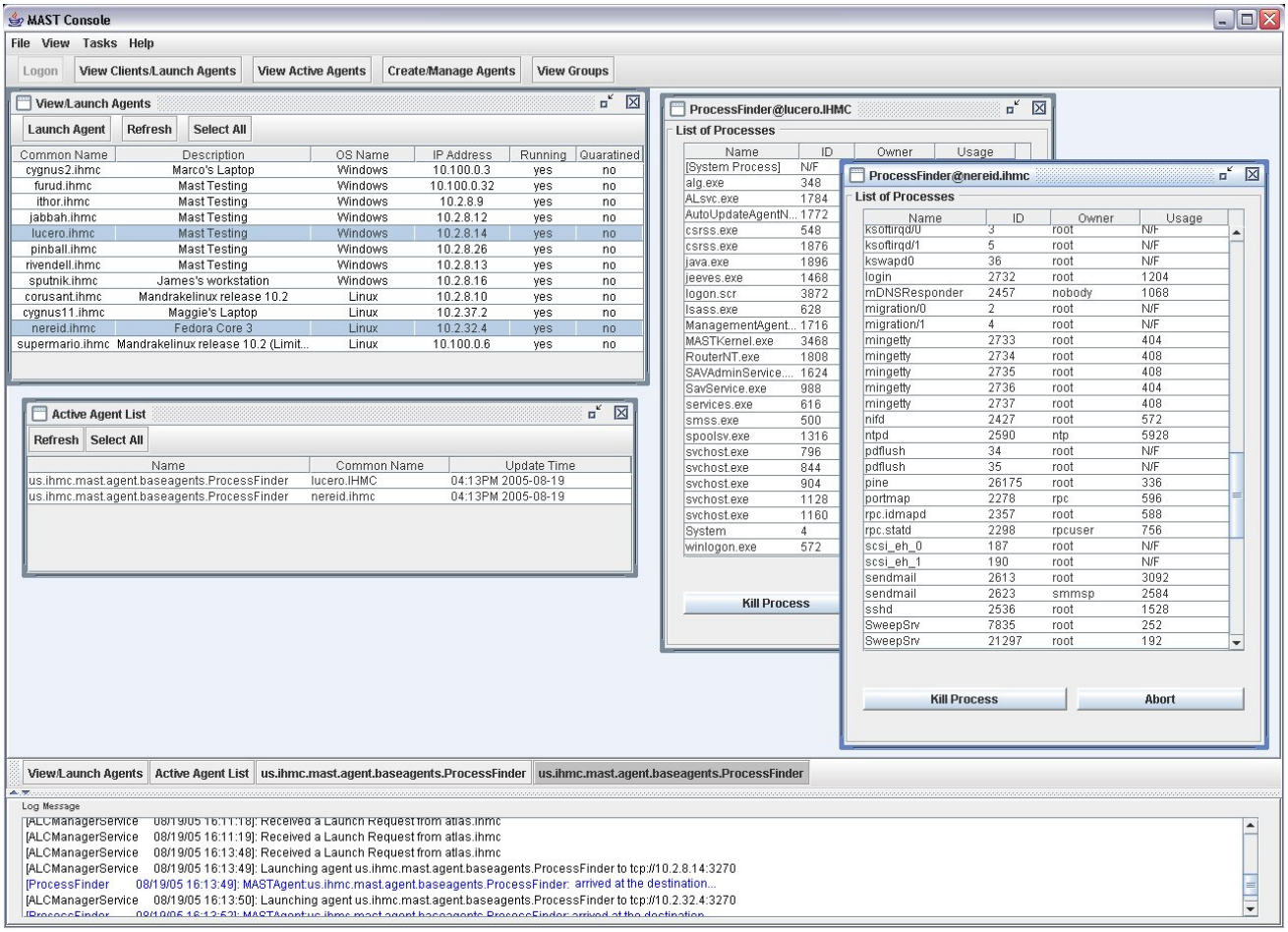

Figure 2 – The MAST console, running two agents with graphical interfaces.

2.1 Security Agents

Security agents in MAST are Java-based classes that can be dispatched to the network to perform arbitrary tasks. The MAST system includes a basic set of agents that perform common system administration tasks. These agents are provided to users as sample code with documentation on how to use system APIs and how to modify or build new agents.

Architecturally, every security agent in MAST extends a base agent class that implements a number of required behaviors. For instance, the base class implements appropriate agent registration, lookup, and termination commands; it also provides the interfaces for logging and sending messages to the console. Security agents can extend these capabilities arbitrarily which allows administrators to build fairly complex agents and agent interfaces if so desired. However, this minimum set of features is always present.

Security agents can be written to work in parallel or sequentially in multiple hosts. If designed for parallel operation, multiple clones of the agent are created and dispatched to a set of machines. Each copy will execute and interact with the user from every machine in parallel. Sequential agents, on the other hand will transverse the list of kernels (or clients) sequentially[2]. The MAST infrastructure will ensure that all specified hosts are visited by the agent. These two types of agents are shown in Figure 3.

As illustrated in Figure 3, a fundamental difference between these two types of agents is that for each task initiated by the user, there will be a single instance of a sequential agent residing (at different times) somewhere in the network, while in the case of parallel agents, there will be multiple instances of the same agent residing in different locations at the same time (and most likely with different internal states). This represents an important factor in determining how the user is expected to interact with each type of agent.

[2] To mitigate the issue of securing the mobile agent against potentially compromised kernels, agents are always required to move back to a server-host (assumed to be trusted) for integrity checking before proceeding to another kernel. This extra intermediate hop is transparent to the user.

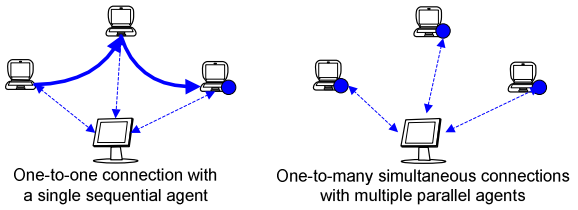

Figure 3 – Sequential and Parallel Agents.

For example, suppose an agent requires a complex graphical interface for interaction. In the case of the sequential agent, every time a query to the user is required, the administrator will be prompted by a single graphical interface. Parallel agents, on the other hand, might create multiple simultaneous interfaces (one from each instance), which does not scale well and is likely to overload the user.

3. THE GROUP MANAGER

The Group Manager (GM) provides a common interface for the system administrator to interact with multiple agents in parallel. GM builds an IRC-like graphical interface that allows the user to simultaneously exchange text messages with any subset of an agent group. The interface is particularly useful for parallel agents.

A Group Manager interface is created for each agent dispatched in parallel to a set of machines. In that case, multiple instances (or clones) of the same agent will be created and launched on the selected hosts. The agents will then form a common functional group, with similar tasks and behaviors at each machine. An example of the Group Manager interface is shown in Figure 4.

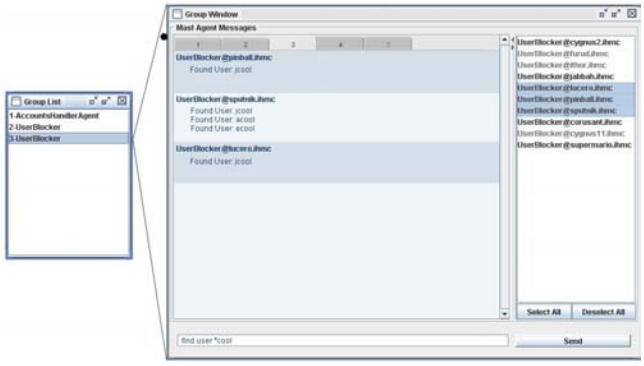

Figure 4 – The Group Manager interface.

The Group Manager interface allows agents to send messages to the user and allows the user to send commands back to the agents. Agent messages can be single or multi-lined and optionally formatted using HTML. Messages are tagged by an agent identifier, which includes the agent name and location. The right side panel in the interface shows the list of agents in a specific group and their current running state (gray indicates that the agent has terminated). Messages are displayed in the left pane in the order they are received by the console. Messages can be filtered on an agent-basis by selecting a specific agent of interest on the right panel, which creates a tabbed window displaying only messages from the selected agent.

Messages typed in the "message" box at the bottom of the dialog box are sent as a text string to all selected agents in parallel. The message will be parsed and handled asynchronously by each copy of the agent in their respective target environments.

Figure 5 shows a snapshot of a group of parallel agents using both interface methods simultaneously. The individual agent dialog boxes are shown on the left side of the console (for only a small number of agents), while the Group Manager interface for the same agents executing the same task is shown on the right side of the console. Although the number of messages is the same, the Group Manager interface facilitates the grouping of events and allows simultaneous instructions to be sent via text messages.

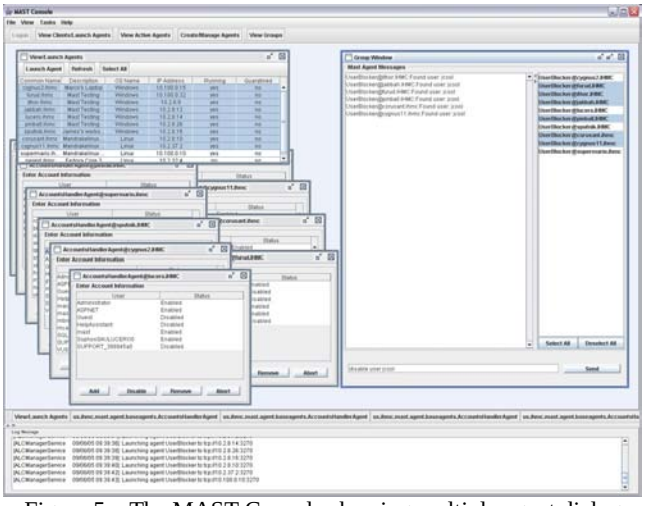

Figure 5 – The MAST Console showing multiple agent dialog messages and the Group Manager interface

4. DISCLOSING COMMANDS

A system administrator in MAST may develop a security agent to perform a number of tasks in response to text commands. The standard way to do that is to parse all text messages received by the agent for recognizable command strings. To facilitate this mechanism, it is necessary for the Group Manager to allow agents to easily present the text commands they can interpret and handle to the administrator. In a network of multiple administrators, this presentation of capabilities is done via a brief description (optionally formatted using HTML). This brief documentation of commands and capabilities is provided by humans (agent designers) to humans (agent users), avoiding requirements for complex formalisms and languages usually necessary for high-level intra-agent communication. This kind of interaction fits nicely within the simple agent framework, allowing administrators to convey a set of capabilities in a simple phrase or brief paragraph describing agent's commands.

To facilitate administrators writing agents that respond to text commands, the base agent class provides a simple mechanism that allows security agents to easily specify text commands to which they will react, the parameters required by the command, and documentation for the command.

In the GM interface for MAST, this is done through a call to the *setCommand()* method. The method takes three string parameters, the first is the text case-insensitive command the user should send

to invoke the desired behavior; the second is the name of the method that should be invoked when such a command is received, and the third is a brief description (documentation) of the method that will be displayed upon user request.

All methods named in the call accept a string array (String[]) as an argument. Every additional space delimitated string that the user types when issuing a command will be set into a string array and relayed to the agent. An example of an agent using the group messaging capability is shown in Figure 6.

A benefit of using this approach over handling the messages directly is that when using the *setCommand()* method, the agent is made aware of its own functionality and the description of its capabilities. This allows the agent's capabilities and description to be presented to the user in detail upon request ("help" command). If messages are handled directly by the agent, the user has no standard way of retrieving that information.

Commands sent by an administrator are pre-parsed and handled by the base agent code. The base agent code checks at that level for general command interpretation, such as "terminate", "help", and "hello".

```
//Agent's constructor
public MyAgent()
{
    //specify command string (vocabulary)
    setCommand ("cmdString1", "method1", "…");
    setCommand ("cmdString2", "method2", "…");
    …
}

//method (handlers) implementation
public void method1 (String[] args) { … }
public void method2 (String[] args) { … }
```

Figure 6 – Defining agent-specific commands

This basic set of commands is handled by all agents (at the base-class level) before being relayed to the administrator's custom implementation (if provided) for special handling. For example, the 'help' command will return to the user the list of base commands and the list of all commands specified through the *setCommand()* method, including their description.

4.1 Command Hierarchy

Agents in MAST are organized in functional groups. Being part of a functional group allows the agent to inherit a set of capabilities that are specific to that particular group.

For instance, the *AccountUtils* group allows the agent to inherit a set of capabilities for system-independent account management such as checking, adding, blocking, or removing a user account, while the *ProcessUtils* group allows the agent to inherit a set of capabilities for managing system processes (e.g. get information about a process, terminate a process, etc.). By creating a new security agent as part of these two functional groups, all the commands in the hierarchy will automatically be available to it. It is important to highlight that the functional hierarchy of agents (which leads to capability inheritance) is not related to class inheritance in Java. When a new MAST agent is created as part of a set of groups with pre-defined capabilities, the actual implementation of the selected capabilities will be copied into the agent's code automatically by the editor. All MAST agents are an extension of the same base class.

As a practical example, consider an agent locating P2P file sharing applications (prohibited by company policies) on corporate workstations. By inheriting the capabilities of *ProcessUtils*, the agent can determine if a P2P file sharing application is running on a host, and then notify the administrator. If the agent also inherits the capabilities of *AccountUtils*, the administrator can direct the agent to report the user associated with the process, and then disable his or her account on that specific machine. In this example, the decisions are made by the system administrator (not the agent), who bases his decisions on information provided by the agent.

5. ILLUSTRATIVE SCENARIO

Consider, for instance, the following scenario. A system administrator of a medium-size network of approximately 200 hosts is notified of a wide-spread propagation of a worm that affects both its Linux and Windows systems. In the example, the information was provided a little too late to prevent the infection so the system will be used for network recovery, as opposed to threat prevention.

The MAST system automatically retrieves the advisory and matches the vulnerable systems with a list of kernels in the network, indicating to the system administrator machines that were vulnerable to the threat and could have been infected by the worm. The worm self-propagates to any vulnerable system and, while running on a machine with an active network, it prevents the application of system patches and immediately re-infects the machine after cleanup. The procedure for removing the worm from the network is to identify infected machines, shutdown their network interfaces, terminate the worm process, locate and remove the worm executable (which is in a well known location), apply system patches, and then re-enable the network interfaces.

Because the procedure involves disconnecting all network interfaces of the machine, the administrative requirements involved in this task are very demanding. System administrators need to physically locate and gain access to all possibly infected hosts (which might include laptops that are currently checked-out for travel or machines that are turned-off in some offices). They will have to address one machine at a time in order to bring the network back to its original state.

In MAST, the worm removal task is simplified by coding the necessary steps for the procedure into a single agent. As agents can operate autonomously and disconnected from the network, it is relatively easy to create an agent that will perform the necessary tasks on each of the target machines. Furthermore, in order to account for special cases (machines that for some reason should not be disabled at the current time) the agent would be interactive. It would autonomously locate the potential threat and prompt the user for the necessary actions. A snippet of the code necessary to perform these tasks in the system is shown in Figure 7. The calls to the *setCommand()* register the "worm_cleanup", "schedule_worm_cleanup", and "abort" request commands. The actual implementation of the *removeWorm()* method performs the necessary tasks required to remove the worm and patch the system. The agent is a local process on each machine and it can operate disconnected from any remote service, which allows it to disable all network interfaces before starting the cleanup process.

All messages sent to or from the agent while the network is inaccessible are queued for later delivery.

```
public class WormRemover extends MastAgent
{
   public WormRemover()
   {
      setCommand ("worm_cleanup", "removeWorm",
                  "Remove the worm");
      setCommand ("schedule_worm_cleanup",
                  "scheduleRemoval",
                  "Schedule worm removal");
      setCommand ("abort", "abortOperation",
                  "Abort operation");
      …
   }

   //Method executed when the agent reaches a host
   public void begin (String[] args)
   {
      …
      if (isWormRunning()) {
         sendMessage ("WormFound!");
      }
   }

   public void removeWorm (String[] args)
   {
      NetUtils.disableNetwork();
      …
   }

   public void scheduleRemoval (String[] args) {…}
   public void abort (String[] args) {…}
   private boolean isWormRunning() {…}
   …
}
```

Figure 7 – WormRemover source code

The agent checks if the system is potentially vulnerable to the threat and reports back to the system administrator through the Group Manager interface. In this particular case the agent sends a message if the worm is found and the administrator can decide whether to remove the worm immediately ("worm_ cleanup"), schedule the worm removal ("schedule_worm_cleanup"), or abort the operation ("abort"). The same agent could have been implemented differently. For example, it could automatically remove the worm if found and notify the administrator only if an exception occurred.

Figure 8 shows the output returned by each agent. In this case only three agents reported that they found the worm. Since one of these machines is currently being used to perform a critical task, the administrator can decide to not remove the worm from this machine. The administrator can remove the worm from the other two hosts by sending the "worm_cleanup" command. For the third host, the administrator can either abort the operation, by sending the "abort" command, or reschedule the worm removal, by sending the "schedule_worm_cleanup" command. In this example, the remaining non-infected hosts could also have been automatically patched by the agent, with no need for human interaction.

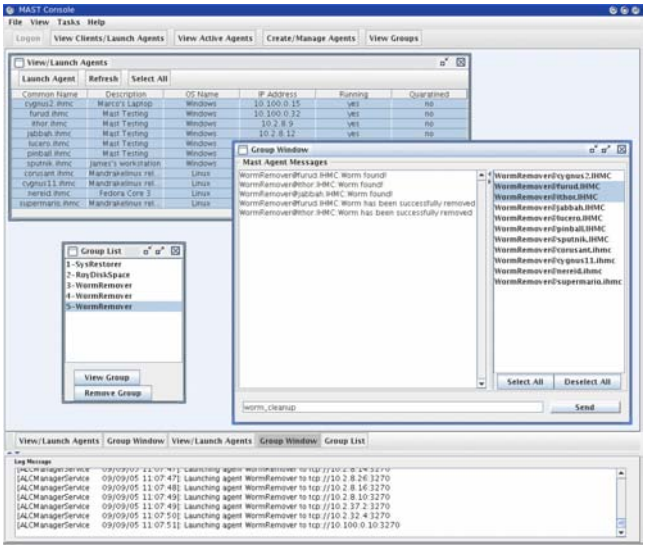

Figure 8 – Group Manager interface for the Worm Remover agent

6. MESSAGE DELIVERY AND OVERHEAD

Although the MAST framework provides facilities for synchronous messaging, in most cases, messaging between agents and users is asynchronous. The messaging service provided by the framework ensures immediate notification if the message fails to be delivered to the target host, or if the addressee is unreachable or nonexistent. The process of sending a message in MAST involves a lookup of the target agent location (usually via SLP[3]) followed by the actual message transmission, which is done through serialization via TCP, eliminating the need for an acknowledgment.

The only guarantee provided by the framework is that the message will be delivered, at some point, to the destination agent, or an exception will be thrown back (synchronously) to the sender. The destination agent is responsible for the actual message handling.

Currently, for a given task, the Group Manager interface does not reduce or increase the number of messages exchanged between the user and agents. That is because MAST does not provide a lower level broadcast (or group multicast) capability so in practice, all messages sent from the user to a group of agents are copied and sent via unicast to each agent.

Message delivery in MAST is reliable from both integrity and non-duplicates perspective. With respect to ordering, there are actually no guarantees of any kind in the current implementation. This could lead to problems if a system administrator sends a sequence of commands to a group of agents without waiting for any feedback (which is in fact optional).

7. EXPERIMENTAL RESULTS

The MAST framework is currently under final experimental evaluation. The experimental results currently available are therefore still preliminary; mostly based on initial tests involving human-subjects.

[3] The Service Location Protocol (SLP) provides a scalable framework for the discovery and selection of network services.

The experimental evaluation of the framework basically consists on having groups of system administrator practitioners perform a number of security-related administrative tasks on a controlled set of Windows and Linux systems. While not using MAST, participants are allowed to use any of the commonly available system management tools such as ssh, netstat, remote desktop, etc. While using MAST, the participants had the option to use a set of basic agents provided with the framework to handle basic system management tasks such as Unix 'ps', 'netstat', etc. Alternatively, participants could also write their own agents for specific tasks.

The test environment was relatively small, containing only 11 workstations under administrator's responsibility. The first phase of tests so far has involved 19 participants, performing tasks with and without the support of the MAST framework and the Group Manager interface.

After the experiment, when asked about the utility of the Group Manger interface to interact with groups of agents in medium to large scale networks, approximately 72% of the participants recognized the interface to be very useful, while 17% percent felt that the interface was useful, but not essential for performing large scale tasks.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a graphical interface for human agent interaction when agents can be grouped and addressed as a functional group, with a well known text vocabulary. The interface was implemented in the MAST framework, with minor extensions to the base agent classes, to support run-time definition and disclosure of an agent's valid commands and vocabulary.

As part of our future work, we plan to implement a more efficient mechanism for group messaging in MAST, improving not only the interface as we propose here, but also the actual traffic associated with user-agent communication.

In the current model, where messages flow from a single user to multiple agents and back, it appears that total and causal ordering or message delivery is unnecessary, however FIFO-order is important and will be the immediate focus for our next version. As we extend the interface to allow direct intra-agent state sharing and commands, further causal ordering requirements will be necessary and will be implemented.

We also plan to explore a different mechanism to automatically create agent groups not only based on code and functional similarity as in our current model, but also based on task synergy.

MAST is freely available for non-profit use. More information about MAST is available at http://mast.ihmc.us.

9. ACKNOWLEDGEMENTS

The MAST project was sponsored by the National Science Foundation (NSF) under the Strategic Technologies for the Internet program, award number 0230927. The authors would also like to acknowledge the collaboration of team members Christopher Eagle, Maggie Breedy and Thomas B. Cowin.

10. REFERENCES

[1] Bieszczad, A., Pagurek, B and White, T.; *Mobile Agents for Network Management*. IEEE Communications Survey Journal, 1998.

[2] Cheikhrouhou, M., Conti, P. and Labetoulle, J.. *Intelligent Agents in Network Management , a State-of-the-Art*., Network and Information Systems Journal, vol.1, n. 1, pp. 9-38, 1998.

[3] Kona, M.K., and C.Z. Xu, *A Framework for Network Management using Mobile Agents*, in proceeding of ICEC 2001

[4] Adhicandra, I., Pattinson, C., Shaghouei, E.. *Using Mobile Agents to Improve Performance on Network Management Operations*. Proceedings of the Post Graduate Network Conference (PGNet 2003), 2003.

[5] Carvalho, M.M, Cowin, T.B., Suri, N. – *MAST – A Mobile Agent-based Security Tool*. – Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics. Orlando, FL, October 2003

[6] Carvalho, M., Cowin, T., Suri, N., Breedy, M., Ford, K., *Using Mobile Agents as Roaming Security Guards to Test and Improve Security of Hosts and Networks* – In Proceedings of the 19th ACM Symposium on Applied Computing. Special Track on Agents, Interactions, Mobility, and Systems (AIMS). March 2004, Nicosia, Cyprus.

[7] Suri, N., Bradshaw, J.M., Breedy, M.R., Groth, P.T., Hill, G.A., Jeffers, R., and Mitrovich, T.S. *An Overview of the NOMADS Mobile Agent System*. Sixth ECOOP Workshop on Mobile Object Systems. (http://cui.unige.ch/~ecoopws/ws00)

[8] Suri, N., Bradshaw, J.M., Breedy, M.R., Groth, P.T., Hill, G.A., and Jeffers, R. *Strong Mobility and Fine-Grained Resource Control in NOMADS.* Proceedings of the 2nd International Symposium on Agents Systems and Applications and the 4th International Symposium on Mobile Agents (ASA/MA 2000). Springer-Verlag.

[9] Cañas, A. J., Hill, G., Carff, R., Suri, N., Lott, J., Eskridge, T., Gómez, G., Arroyo, M., & Carvajal, R. (2004). CmapTools: A Knowledge Modeling and Sharing Environment. In A. J. Cañas, J. D. Novak & F. M. González (Eds.), *Concept Maps: Theory, Methodology, Technology, Proceedings of the 1st International Conference on Concept Mapping*. Pamplona, Spain: Universidad Pública de Navarra.

[10] Novak, D.B. Gowin, *Learning How to Learn*. Cambridge University Press (1984)