The RETSINA Communicator*

Onn Shehory†
IBM Research Lab in Haifa
The Tel Aviv Site
2 Weizmann St., Tel Aviv, 61336 Israel
onn@il.ibm.com

Katia Sycara
The Robotics Institute
Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213 USA
katia@ri.cmu.edu

1. INTRODUCTION

We are developing the RETSINA[1] Multi-Agent System (MAS) [4] in which multiple agents receive goals from users and other agents. Since RETSINA implementations are deployed in real-world, distributed, open environments, it is necessary that agents be able to communicate both among themselves and with other entities, to enhance coordination and cooperation and increase system performance. Yet, communication based on client/server architecture is insufficient for the needs of agents, as agents may be both service providers and service consumers, sometime simultaneously. In addition, to support some aspects of interoperability, agents must be able to communicate using standard Agent Communication Languages (ACLs). Thus, there is a need for a more expressive means of communication. As part of our agent architecture, we have developed a communication component - the RETSINA communicator, that provides these requirements, and more. It supports concurrent connections to other agents in client, server, and peer-to-peer fashions; it supports the KQML [2] ACL and can easily be adjusted to FIPA [1] and other ACLs; it is designed to be easily re-used in a plug-in manner; it is implemented in several programming languages and was tested on several different platforms. The RETSINA communicator is loosely tied to agents, and can be used in any application where similar communication needs arise.

RETSINA is an open MAS that provides infrastructure for different types of agents. Each RETSINA agent [4] is composed of four autonomous functional modules: a communicator, a planner [3], a scheduler and an execution monitor. The communicator module receives requests from users or other agents in KQML format and transforms these requests into goals. It also sends out requests and replies. The planner module transforms goals into plans that solve the goals. Executable actions in the plans are scheduled for execution by the scheduler module. Execution and monitoring of the actions is performed by the execution monitor module. Some actions may need, and can invoke, communication. This is performed by the action directly approaching the communicator and using its public methods.

[1]REusable Task Structure based Intelligent Network Agents.

*This work was supported in part by ARPA/Rome Labs contract F30602-93-C-0241 & ARL contract DAAL0197K0135. We thank Dirk Kalp for his invaluable contribution.

†At the time of conducting this research, Onn Shehory was affiliated with Carnegie Mellon University.

2. THE COMMUNICATION MODULE

The RETSINA communicator provides an abstraction that supports peer to peer communication between agents based on the names of the agents. The communication is in the form of ASCII messages transmitted on TCP/IP sockets. The communicator provides an API specification for establishing connections to external agents and for several forms of asynchronous communication for the sending and receiving of messages with options for blocking or non-blocking communication. There is also a synchronous interface. Connection setup and management is handled transparently within the communicator. The mapping of agent names to their host and port addresses is done by the communicator through the RETSINA Agent Name Server (ANS). The ANS supports distributed agent name registration and lookup.

The communicator does not support complex message protocols or specific message formats. It simply transmits messages, leaving message protocols to the higher level modules in the agent. With respect to message formats, the communicator sets forth minimal requirements by defining an interface that provides the abstraction it needs to handle messages that come in from external agents and a separate interface for the messages that the agent, in which the communicator is incorporated, wishes to send out. These interfaces, `ExternalMsg` and `InternalMsg` respectively, define a small number of basic message fields deemed essential for inter-agent communication along with accessor methods for extracting and setting message fields. We define two different interfaces since the communicator has slightly different needs at each end of the communication management and for greater flexibility and portability of agents into different agent communities. For example an agent developed in a community that uses KQML-based communication can be easily ported to a FIPA-based community by supplying a new module that implements the `ExternalMsg` interface to support FIPA formats. The communicator deals

with messages using the abstractions of `InternalMsg` and `ExternalMsg` and thus is divorced from particular formats.

3. IMPLEMENTATION DETAILS IN BRIEF

We have separated out the `CommunicationInterface` that defines the abstract methods for inter-agent communication. The `Communicator` implements the methods of this interface to provide the API through which an agent can establish and carry out communication with external agents. The term agent is used loosely here, referring to any agent, client/server program, or other application program that incorporates the `Communicator`. A quick reference to the main public methods of the `Communicator` API is provided below.

At initiation, an agent's `Communicator` registers the agent's name, host and port with an ANS. To establish communication with another agent, the `Communicator` queries the ANS with that other agent's name in order to obtain its host and port. The `Communicator` then uses the host and port to establish a connection to the external agent. The endpoints of a connection are sockets, one in the agent and one in the external agent. Creation and management of the agent's connections is handled transparently by the `Communicator`. The agent does not deal directly with the sockets or connections. Instead, through the `Communicator` API, the agent can obtain a `ConnectionDescriptor` object that is a reference to a connection with another agent.

To obtain a `ConnectionDescriptor`, the agent uses either `openConnection()` or `openExclusiveConnection()`. The only argument required is the name of the agent to which a connection is desired. Given the name, the `Communicator` will lookup the host and port of the external agent at an ANS, create a socket and connect it to the socket at the advertised host and port of the external agent. It is possible for the agent to have multiple connections to the same external agent. For this, the agent uses `openExclusiveConnection()` to get a descriptor to a new connection. Non-exclusive connections are obtained via `openConnection()`. This still assigns a new descriptor but it may point to an existing connection to the particular external agent. The `Communicator` keeps a reference count on connections that have multiple descriptors to assure that a connection is not destroyed until each of its descriptors is disposed of (via `closeConnection()`).

To send messages, the agent uses `sendMsg()` or `queueMsg()`. The former is used for urgent and priority messages, with little error handling, whereas the latter uses retry/recovery strategies to handle queued messages until they are successfully sent to the intended party. The `sendMsg()` and `queueMsg()` methods provide an asynchronous interface for the *message sending* part of the agent communication. Although asynchronous communication is more useful in open MAS, the API also provides a method, `sendMsgAndGetReply()`, for synchronous communication, which sends a message to the external agent and then blocks the calling thread waiting for the reply message from that agent.

The client and server roles are duals of each other. A peer-to-peer communication model is one in which the agent performs both of these roles and in accordance with the agents needs. An agent will usually be willing to provide some service and take on the server role in listening for and accepting new connections initiated by the external clients. An agent will also usually need to make requests itself for the services of other agents, taking on the client role of initiating a connection to a server or other external agent. In the client role, initiating a connection is done by calling one of the two `Communicator` methods for opening a connection. In the server role, the agent needs to discover when an external agent operating in the client role initiates a connection. The `Communicator` handles this connection creation passively in response to the external agent's connection initiation.

The control of connections is determined by connection ownership. An agent's owned connections are those explicitly created by the agent in the client role. The agent's un-owned connections are those that are passively created by the agent in the server role. A `closeConnection()` operation can be performed only by the connection owner. It is likely that an agent be multi-threaded, some threads handling the owned connections, the client role, while other threads handling the un-owned connections, the server role.

At termination, the agent calls `refuseNewConnections()` to prevent any new client connections from being accepted by the agent. It then should call `flushMsgsAndStop()` in the API to flush any output messages queued and stop all `Communicator` threads. Following this, `unRegisterWithANS()` should be called in the `Communicator` to remove the registration of the agent with the ANS.

4. CONCLUSION

The RETSINA communicator was implemented in Java, Python, C++, and Lisp. Besides our RETSINA agents, we have incorporated it into two other agent interface applications and a server. The `Communicator` supports multiple open connections between two agents. It also provides an interface for performing logging of message traffic. In future research, we intend to extend this logging facility and develop an abstraction of the use of the ANS so that some other agent name service facility could be substituted if desired. In summary, the RETSINA communicator is a reusable component for agent communication. It can easily be plugged into new agent and applications, as well as being adjusted to different ACLs. Most importantly, it supports communication in client, server and peer-to-peer fashions, possibly all at the same time. These attributes made the RETSINA communicator a useful, widely used, generic component for agents and application in open dynamic MAS.

5. REFERENCES

[1] FIPA –*Foundation for Intelligent Physical Agents.* http://www.fipa.org

[2] T. Finin, R. Fritzon, D. McKay and R. McEntire. KQML – A Language for Knowledge and Information Exchange. *In Proc. 13th Intl. DAI Wshp, Seattle, 1994.*

[3] M. Paolucci, D. Kalp, A. Pannu, O. Shehory and K. Sycara. A planning component for RETSINA agents. *ATAL-99, 1999, Orlando, Florida.*

[4] K. Sycara, K. Decker, A. Pannu, M. Williamson, and D. Zeng. Distributed intelligent agents. *IEEE Expert – Intelligent Systems and Their Applications*, 11(6):36–45, 1996.