MCC Technical Report Number ACT-AI-378-89

Distributed Reasoning among Expert Systems

Natraj Arni and Michael N. Huhns

October 1989

MCC Nonconfidential

Abstract

This report describes the process and result of converting a conventional expert system shell to one that enables distributed reasoning. The resultant shell, *Antares*, can be used to construct individual expert systems that communicate and cooperate with each other in solving problems.

Microelectronics and Computer Technology Corporation
Artificial Intelligence Laboratory
3500 West Balcones Center Drive
Austin, TX 78759-6509
(512) 338-3651
huhns@MCC.COM

1 Introduction

After more than a decade of successful exploitation of knowledge-based systems, we have hit a barrier in the size and complexity of the problems that can be solved with current technology. Larger problems cannot be solved because the requisite systems are too complex to build and maintain. The system described herein will make it possible to solve larger, more complex reasoning problems. Our strategy is to provide novel technology for **distributed reasoning** among multiple knowledge-based systems. Distributed reasoning enables the systems to cooperate in problem solving, negotiating conflicts as necessary. Users can then exploit a *divide-and-conquer* approach to development: they will be able to build smaller, more manageable knowledge-based systems that *cooperate* in solving complex problems. These smaller systems can also be *reused* in different combinations for solving other problems in the future.

In this report we describe the primitives and features that a tool for building knowledge-based systems needs in order to support distributed reasoning. The following is a typical scenario for the use of such a tool: there is a loosely-coupled network of experts, each being an expert in a particular field, and there is a problem that these experts must solve that is beyond the capabilities of any one of the experts. More specifically, there exists a number of identically-structured[1] knowledge-based systems (agents), each having its own domain specific knowledge. There may or may not be a *user* associated with each one of these agents. A problem to be solved, or a goal to be satisfied, is given to one of these agents by a user. This agent does one of the following three things:

1. Solves the problem or satisfies the goal itself, using local knowledge and inherent reasoning mechanisms.

2. Decomposes the problem or goal into subgoals, distributes each one of these subgoals to an appropriate agent, and then integrates the solutions returned by the other agents.

3. *Tries* to solve the problem itself, but discovers a subproblem it cannot solve. It distributes that subproblem to another agent, and then incorporates that agent's solution into its own.

[1] At least for now.

In any case, the agent reports the final solution to the user.

The features required for distributed reasoning include a communication channel among the agents, a communication protocol for exchanging goals and solutions on this channel, and a reason-maintenance system that enables globally-coherent solutions to be achieved. We have incorporated these features into Antares, a full-featured shell for building knowledge-based systems. Antares supports a forward and backward rule-based inferencing mechanism, a frame system for knowledge representation, a justification-based truth-maintenance system [Doyle 1979, Petrie 1989], and a high-performance implementation derived from the Warren Abstract Machine [Warren 1983], as well as the distributed reasoning mechanisms.

2 Model for Distributed Reasoning

Antares' model for distributed reasoning among multiple knowledge-based systems is based on the interactions between a user and a single, conventional knowledge-based system. In this model, a user can query and make assertions to the knowledge-based system, to which the system responds with either an answer or an acknowledgement, as appropriate. As a side-effect of forward or backward inferencing, the system can independently inform or make a request of a user, who must answer all requests. This model permits interactions to be either user-initiated or system-initiated.

Antares extends this model to the multiple agent case by enabling system-initiated interactions to be directed to another agent, as well as to a local user. For this purpose, a destination field is added to each query or assertion. A value for this field, i.e., the name of an agent who should receive the query or assertion, can then be the subject of a reasoning process in the originating agent. Since an assertion can now come from several sources, its justification must be modified to record a source. An agent must also record the agents to whom it sent an assertion, so that it can inform them in case its belief in that assertion ever changes.

3 System Architecture

The agents exist in a fully distributed environment. Each agent can make queries of other agents and also serve as a knowledge base for other agents. That is, each agent works in a *Client-Server* mode. It is a *Client* when it requests other agents to solve a problem or satisfy a goal. It is a *Server* when it satisfies goals for other agents. Each agent has the following state variables associated with it:

1. System-Lock (**LOCKED OPEN**)
2. Input-Buffer
3. Status (**WAIT BUSY FREE**)
4. Input-Buffer-Lock
5. Authorization-Lookup-Table
6. Priority-List

When an agent queries another agent, it can either go into a WAIT state until it receives a result or dispatch the request and carry on with other chores. The first choice refers to a *serial* way of solving problems and the second one refers to a *parallel* way of solving problems. In the current implementation, the agents operate serially. When an agent is in the WAIT state, it should not permit other agents to perform either destructive operations on it or make queries of it, as this may lead to a potential deadlock. This deadlock issue is resolved by a time-out mechanism. Hence, a proof attempt on an agent that is either in a WAIT or a BUSY state will *fail*. An agent also has multiple channels open to some or all other agents. Agents can also assert facts on remote agents. In Antares, this is allowed only when an agent obtains a lock on another agent whose database it is trying to alter. The reason for locking is obvious: as long as an agent holds a lock on another agent, no other agent can either query or assert facts on the locked agent. This is necessary to keep the database consistent and the query processing deterministic. A user can utilize the windowing system to view and edit the knowledge base of a remote agent from the user's terminal.

The following are the salient issues in distributed reasoning:

- Remote queries
- Remote justifications (to enable the implementation of a distributed JTMS [Mason and Johnson 1989])
- Negotiation
- Multiple users

We discuss each of these issues briefly, giving examples as we go along. This is not an exhaustive list and there are also some unresolved issues.

3.1 Remote Queries

This is one of the most basic operations in Antares. An agent or user can make queries of other agents. The queries could be either simple queries, i.e., queries at the user-interface level, or proof attempts of a literal of a rule at a remote agent. The goal to be proved is sent to the remote host (the *server*) along with a *Client Id.* This way we register a request at the *Server.* If the server is *free* and its lock is *open*, an attempt is made to prove this goal. If a succesful proof is obtained, a datum [Proteus 1989] of the type *Server-Datum* is created. This datum has the justifications for the proof of the goal. This datum also has a field to store the Client Id. For this datum, a tag is generated. The datum thus generated is stored in a local associative table, the tag being the associative key. Note that this datum participates "normally" in the local TMS. For each succesful proof, we perform the operations described above. Finally all the bindings and their corresponding datum tags are sent back to the client. An example of a query to and a reply from "Agent1" is shown below.

```
Query: (remote-query ''Agent1'' (foo ?x ?y))

Receive: (''((:bindings ((?x 10) (?y 20)))
    (:datum-tag Server_Tag_Agent1_65624)
    (:status 'IN)
    (:server ''Agent1''))
   ((:bindings ((?x a) (?y b)))
    (:datum-tag Server_Tag_Agent1_65623)
```

```
(:status 'IN)
(:server ''Agent1''))
```

The predicate, **remote-query**, translates to a low level query to the remote agent with the goal and the client as the arguments.

The Client now creates a datum of the type *Client-Datum* for each proof received from the server. This is an "image" of the server datum. Information about the server is stored in this datum. This datum is given a remote justification, which is a special kind of justification [Proteus 1989]. Finally, this datum is stored in a local associative table at the client with the tag being the associative key. Note that this datum participates in the client's TMS. Hence, the query could be a part of a rule. This datum is a special kind of datum, whose status (IN/OUT) depends not only on local justifications, but also on proofs from remote agents. This is the justification for the above bindings. Finally, this datum is assigned the status IN.

3.2 Remote Justifications and Distributed TMS

Remote data and justifications are created as described in the previous section. When the remote agent changes its belief about a datum, i.e., changes its belief status, and this datum happens to be of type Server-Datum, then the corresponding Client, which is responsible for the creation of this datum, is notified. At the receipt of this message at the client, the "image" datum at the client is accessed. It is this datum that needs to be acted upon. If the transition is from an OUT to an IN, a new remote justification is created at the Client and is pushed in to the justifications of the client datum and the status of the datum is made IN. If the transition is from an IN to an OUT, the client datum is made OUT by an ERASE opertaion. Again, the justification for the ERASE is a newly created remote justification. The information about the agents causing the above changes is stored in the remote justifications. The Client also has an option of ignoring the messages from its servers, when the remote data it depends on change their status. This can easily be set by a flag.

Note that the above discussion on remote justifications is necessary only for proofs whose justifications are needed. This may exclude simple user-level queries, whose proof tree (or justification information) is transient. An

example where the justification is needed is remote proof attempts on one or more antecedents of a forward rule. These remote justifications will have to be entered in the justification of any assertion of the consequents of this forward rule.

3.3 Example

Suppose there are two agents called Agent1 and Agent2, and Agent1 would like some help from Agent2 in choosing a stock to buy. Agent1's query to Agent2 about this initiates the exchange of information depicted in Figure 1. In this example, Agent1 knows what it can afford, but it does not have any knowledge about what stocks it would be wise to buy. It does know, however, that Agent2 knows about this problem. So it attempts to solve this subgoal by sending a query to Agent2.

Agent2 does have some general knowledge about what stocks to recommend and about what stocks it can afford. It is interesting to note that Agent1 and Agent2 have different and even conflicting knowledge, e.g., Agent1 believes XCorp is affordable, while Agent2 does not. But because Agent1 and Agent2 do not choose to interrogate each other on this issue, their differences of belief do not matter. These agents represent two different viewpoints about the world that are not easily modeled by a single, conventional expert system. Further, we see that two agents can cooperate to solve a problem even if their beliefs are not identical.

When Agent2 receives Agent1's query, it attempts to find an answer by applying its rules. It succeeds and binds ?x to XCorp. It also creates a server datum for this conclusion. This datum (labeled Server-Agent2-12 in the figure) contains the conclusion, the justification for the conclusion (namely rules 1 and 2), and a pointer to the agent to whom this conclusion was reported (in this case Agent1). Agent2 then reports its conclusion to Agent1.

Now Agent1 can continue its attempt to apply its rule 1. It succeeds at that, with ?x bound to XCorp, and it generates the assertion (buy XCorp). It justifies this with rule 1, fact 3 (that XCorp is affordable), and a client datum (Client 5) that records the fact that Agent2 recommended XCorp. No further interaction between Agent1 and Agent2 is necessary at this point.

Agent2 does, however, retain a commitment to notify Agent1 if its belief in the affordability of XCorp changes. The client and server data that have

been constructed provide a means of doing that in case the need arises. In that case, Agent1 will retract its decision to buy XCorp.

3.4 Negotiation

There is no requirement for two agents to agree completely. That is, each agent maintains a locally consistent set of beliefs using its own justification-based truth-maintenance system, but global consistency among the agents is not required. However, when two or more agents disagree about belief in a datum *and when this disagreement is encountered during problem solving*, then negotiation among the agents will ensue to resolve the disagreement. The negotiation is necessary to ensure that the global solutions to the problems posed to the agents are coherent.

The negotiation procedure involves an exchange of justifications among the agents. For example, if there is a disagreement about belief in datum D_1, then the agents exchange the justifications for their belief or disbelief in D_1. The agents then compare these justifications. If there is a disagreement about belief in any datum in the justifications, then the justifications for this disputed datum are exchanged, and the process recurses. It is expected, but not yet proven, that resolution of all disagreements among two agents in a consistent world will be achieved by one agent informing another of some explicit knowledge that the other is missing. The negotiation process is an active area of research at MCC and will be described further in a forthcoming technical report.

3.5 Multiple Users

Antares allows multiple users, but, for safety, there can be only one user at any one time; the locking mechanism described in Section 3.1 prevents this. Alternatively, Antares may someday be made reentrant for true multitasking.

3.6 Protocol for Message Passing

On the Symbolics, we use the Chaosnet [Symbolics 1988] as the medium of communication. A protocol, called Remote-Eval-Server, is then defined on this medium. This gives us the ability to send Lisp forms across the network,

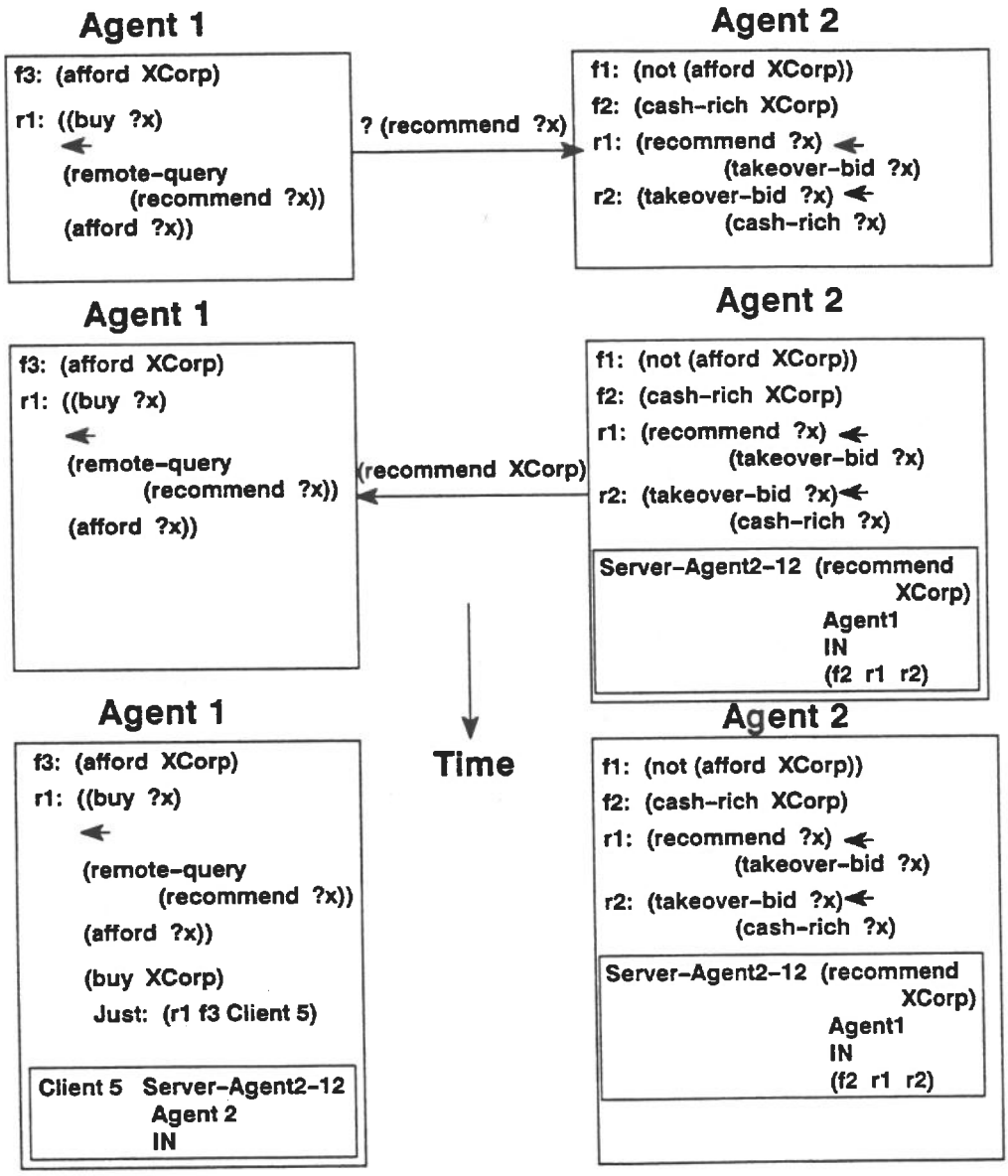

Figure 1: The steps involved in processing a remote query. Note the justification tables built by both the server and the client agents.

packed in a string. These strings are unpacked at a remote machine, decoded and executed. The values are returned using a similar method.

For a UNIX environment, we use the Inter Process Communication (IPC) primitives [UNIX]. Multiple agents communicate with each other using sockets and pipes. The protocol is TCP/IP.

Messages between agents are passed using strings. These strings contain TOKENS separating various fields in the message. It is assumed that all the agents understand the same language and hence can parse the message.

3.7 Global Namespace Server

One agent will be designated as the global namespace server; it will maintain the identities and locations of all of the agents.

4 System Configuration and Use

For Symbolics workstations, the Host object on each workstation must be edited using the Namespace Editor to add a service called

```
Service: REMOTE-EVAL CHAOS REMOTE-EVAL-SERVER
```

Once this is done, a user can define a workstation as both a Server and a Client, using the following forms:

```
;;*************************************
;;; This defines a server

(net:define-server :remote-eval-server
    (:medium :byte-stream :stream stream)
   (with-standard-io-environment
     (let ((input-form (read stream)))
       (print (eval input-form) stream))))

;;*************************************
```

```
;;; This defines a client

(net::define-protocol :remote-eval-server
      (:remote-eval :byte-stream)
  (:invoke-with-stream-and-close (stream form)
    (with-standard-io-environment
      (print form stream)
      (force-output stream)
      (read-from-string
(with-output-to-string (string-stream)
  (stream-copy-until-eof stream string-stream)
  )))))


;;; This defines a contact name for the Protocol.

(chaos:add-contact-name-for-protocol :remote-eval-server "R-EVAL")
```

For the UNIX environment, each agent starts up a socket pair with a unique name or Id. It is assumed that other agents, wishing to communicate with any other agents, know about their socket names.

4.1 Library

The file called dai.prot should be consulted at start-up time. It should be in either the user's home directory or /usr/lib/antares/dai.prot. This file contains the following two rules:

```
(assert
  ((remote-prove ?host ?goal)
   <--
   (is ?all-proofs (pi:host-prove ?host ?goal))
   (element (?goal ?datum) ?all-proofs)
   (is ?exists (pi::datum-p ?datum))
   (push-cl ?datum))
```

```
  ((remote-assert ?host ?pattern)
    -->
   (is ?success (pi:host-assert ?host ?pattern)))
)
```

4.2 Syntax

The syntax for remote queries is straightforward. The following is the general form of a remote query:

```
? (remote-prove <hostname> <goal>)
```

The syntax for a remote assert is

```
? (remote-assert <hostname> <assertion>)
```

5 Conclusions

During the next five years, the Reasoning Architectures group at MCC will be developing new technologies to enable **distributed** reasoning in AI systems. It is our firm belief that such technology will be crucial for the future effectiveness of reasoning tools. This belief is motivated by a number of observations:

- Applications requiring an AI approach are increasingly large and diverse, yet existing reasoning tools work well only for small, specialized problems. New tools are required that allow modular development of knowledge bases.
- The world is distributed. Information sources abound throughout our environment; there is no way for a single reasoning agent to effectively manipulate all of the information available. Reasoning tools must be

able to communicate and cooperate with one another, just as human experts do.

- Once an expert system is developed with today's shells, it becomes an isolated island of knowledge. It cannot easily be reused in other, similar problem domains. Because of this, the cost of engineering and maintaining multiple expert systems is already becoming noticeably high in the business community, where AI applications are increasingly commonplace.

Antares is a first step toward cooperative distributed problem solving among multiple agents. It provides the low-level communication and reasoning primitives necessary for beneficial agent interactions, but it does not guarantee successful and efficient cooperation. The next steps will require increased intelligence and capabilities for each agent, resulting in more sophisticated agent interactions occurring at a higher level. We are providing these capabilities through our research.

References

[Doyle 1979] J. Doyle, "A Truth Maintenance System," *Artificial Intelligence*, vol. 12, no. 3, 1979, pp. 231–272.

[Gasser and Huhns 1989] Les Gasser and Michael N. Huhns, eds., *Distributed Artificial Intelligence, Volume II*, Pitman Publishing, London, 1989.

[Huhns 1987] Michael N. Huhns, ed., *Distributed Artificial Intelligence*, Pitman Publishing, London, 1987.

[Mason and Johnson 1989] Cindy L. Mason and Rowland R. Johnson, "DATMS: A Framework for Distributed Assumption Based Reasoning," in [Gasser and Huhns 1989], pp. 293–317.

[Moore 1980] R. Moore, "Reasoning about Knowledge and Action," Technical Note 191, SRI International, 1980.

[Petrie *et al.* 1986] C. J. Petrie, D. M. Russinoff, and D. D. Steiner, "Proteus: A Default Reasoning Perspective," *Proceedings 5th Generation Conference*, National Institute for Software, October 1986.

[Petrie 1989] C. J. Petrie, "Reason Maintenance in Expert Systems," MCC Technical Report No. ACA-AI-021-89, Microelectronics and Computer Technology Corporation, Austin, TX, February 1989.

[Warren 1983] David H. D. Warren, "An Abstract Prolog Instruction Set," SRI Technical Note 309, SRI International, October 1983.

[Symbolics 1988] "Networks," *Symbolics Users Guide*, vol. 9, Symbolics, Inc., Cambridge, MA, 1988.

[Proteus 1989] Natraj Arni, et al., "Proteus 3: A System Description," MCC Technical Report No. ACT-AI-226-89-Q, Microelectronics and Computer Technology Corporation, Austin, TX, June 1989.

[UNIX] "UNIX Programmers Manual," University of California, Berkeley, CA, 1984.