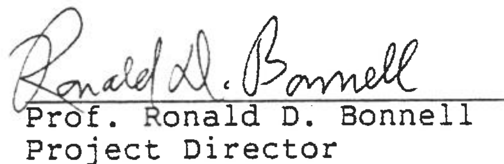
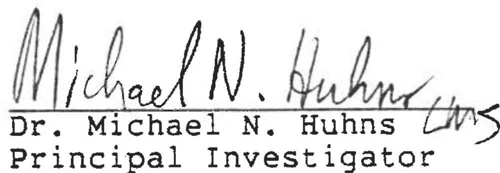
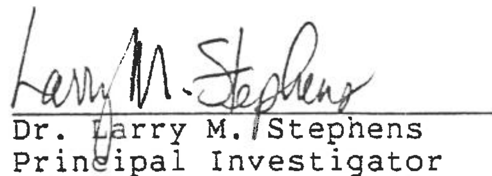
A Final Report to
Office Systems Division
NCR Corporation
Columbia, South Carolina
June 7, 1985

PART I: IMPLEMENTATION OF A DISTRIBUTED EXPERT SYSTEM FOR THE INTELLIGENT FILING SYSTEM (IFS)


Prof. Ronald D. Bonnell
Project Director


Dr. Michael N. Huhns
Principal Investigator


Dr. Larry M. Stephens
Principal Investigator

Center for Machine Intelligence
Department of Electrical and Computer Engineering
University of South Carolina
Columbia, SC 29208
(803) 777-4195

FINAL REPORT, PART I:
IMPLEMENTATION OF A DISTRIBUTED EXPERT SYSTEM
FOR THE INTELLIGENT FILING SYSTEM (IFS)

TABLE OF CONTENTS

I. INTRODUCTION 1

II. NETWORK SERVER 4

- A. LISP Implementation 4
- B. The C Implementation 9

III. QUERY PLANNER 11

- A. Structure of the Query Blackboard 13
- B. Query Planning 13
- C. Database Interface 13
- D. Query Structure 14

IV. RESPONSE PLANNER 16

- A. Response Blackboard Structure 16
- B. Global Variables for Response Planning 17
- C. Response Planner Operation 18

APPENDIX. Program Listings 20

A DISTRIBUTED EXPERT SYSTEM APPROACH TO THE INTELLIGENT FILING SYSTEM (IFS)

I. INTRODUCTION

This report discusses the implementation of the designs for the Network Planner, Query Planner, Response Planner, Learning Subsystem, and Data Model. The Network Planner is responsible for establishing network connections among the MINDS systems and organizing queries and their responses into the proper messaging format. In addition, it will maintain information on the status of the other nodes in the network in order to facilitate communications. Figure 1 shows the relationships among MINDS and the other IFS subsystems.

The Query Planner is the essence of the MINDS system. When a query is received from the user interface, a sequence of actions is generated to satisfy the query. This sequence typically involves the retrieval of information from the databases of document surrogates and metaknowledge, the accessing of files via the OS/FMS, the communication of messages through the Network Server, the processing of errors, and the generation of responses to the user interface. This sequence, or plan, will be constructed for each allowable input from the user interface.

The Response Planner is responsible for merging partial responses to queries into a final response which is transmitted to the user interface. It must keep track of which nodes are being queried and which have responded. Metaknowledge received from other nodes in the network is collapsed by the response planner into an updated metaknowledge table. The Response Planner also cooperates with the Query Planner by informing it of other nodes to query. This is accomplished by examining local metaknowledge and comparing it to a threshold value for query generation.

The Learning Subsystem incorporates heuristics for revising metaknowledge. The heuristics implemented are those governing updating local metaknowledge based on metaknowledge received from other nodes and those for reducing to zero the metaknowledge about a user who, when queried, supplies no documents about a particular keyword. Other heuristics can be incrementally added to the system as desired. The Learning Subsystem represents the most significant research area in the MINDS project and is sufficiently novel to bring attention to the intelligent filing system in the open literature.

The data model for MINDS has been established and is implemented using the MISTRESS 32 database management system. Tables for document surrogates and metaknowledge are included.

The above subsystems are implemented in Franz Lisp and C on a network of three SUN-2 workstations. In particular, this prototype includes the following modules:

1. Network Planner

2. Query Planner (with command parser)

3. Learning Subsystem

4. Database

5. Query and Response Blackboards

6. Response Generator

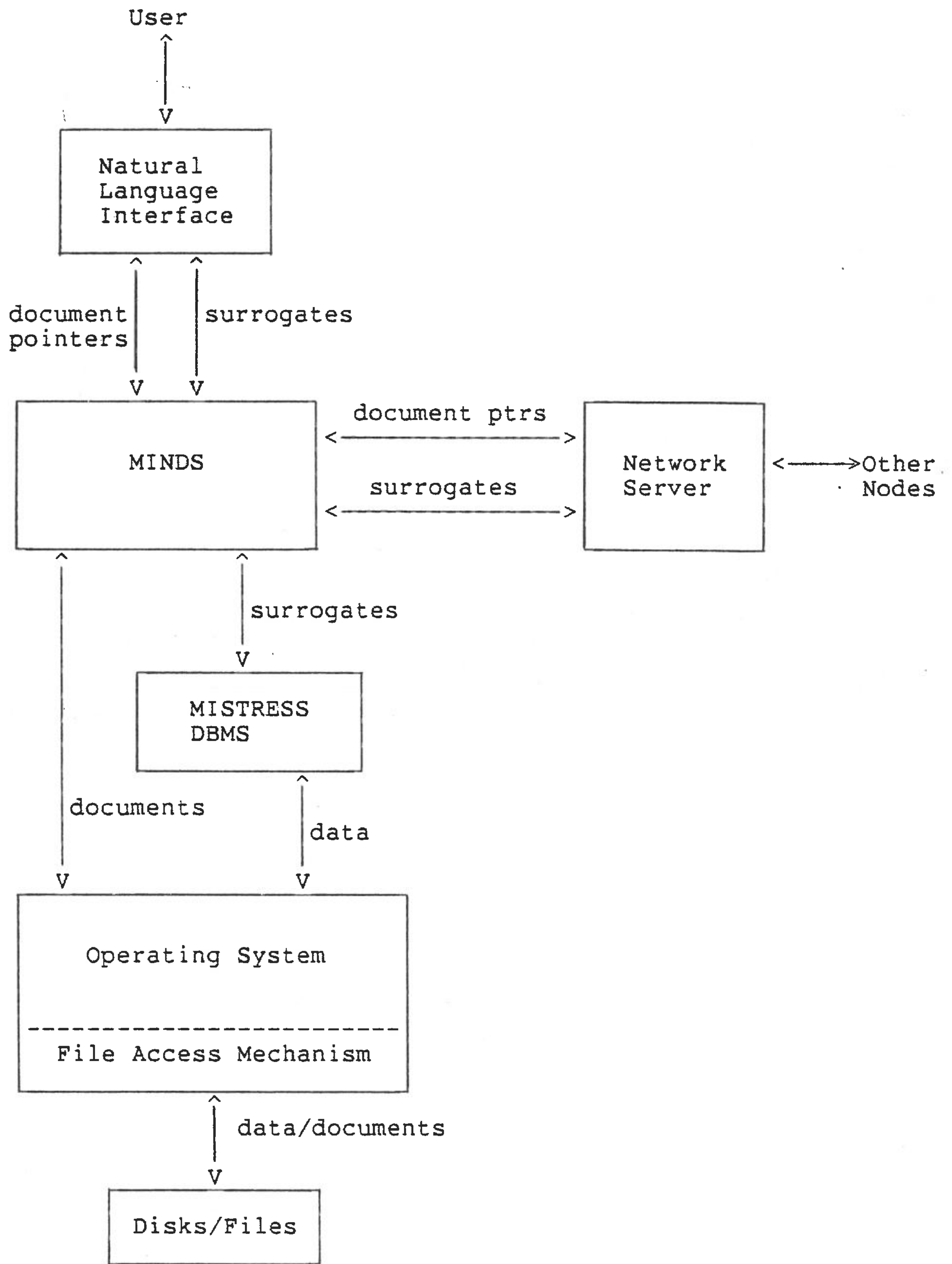


Figure 1. The Relationship of MINDS to other IFS Components

II. The Network Server

The Network Server is designed to meet the specific needs of the MINDS project. The result is a communications package for implementing a fully-connected network. In a fully-connected network each node has a direct link to every other node. This type of configuration is rarely used commercially due to the complexity and expense. It was chosen for the prototype since it consists of only three nodes, each node can easily address all other nodes, and each carries an equal burden for network communications. The Server consists of two major parts: A LISP portion which is embedded in the MINDS program and a C portion which implements the basic internode communication primitives and runs concurrently as a separate process.

A. LISP Implementation

The following section describes the LISP implemented portion of the Network Server. The C process which supports the communication primitives (recv and send) is described in the next section. These two functions are written in LISP. The goal of the Server is to make all communications outside of a given node transparent to the other modules. The Query Planner and the Response Planner determine the exact form of the data structures at each node and the Server complies by using whatever data structures the designers of the Query Planner and Response Planner have chosen. If the local data structures are changed, only minor routing changes in the LISP portion of the Server and no changes to the C portion of the Server are required. The other modules place information only in the local data structures. For instance, the Query Planner places a query which should be sent to another node in the 'to_remote' buffer; then the Server sends the query out. The Server at the remote node places the incoming query in the correct buffer at the remote node. The Server is the only module which knows about physical device names. It is sufficient for the other modules to refer to nodes only by their logical name, such as user_1.

Within a node the Server must take care of the following two situations: 1) It must receive all incoming packets from the sockets and place them in the correct data structure and do the required updating associated with that packet, and 2) It must transmit all eligible packets from the local data structures to the network by sending them to the appropriate socket.

We first examine the role of the Server for receiving information. All information comes into a node through the sockets. There are two sockets for the two other nodes in the network and one socket which connects to the user interface. The sockets are named user1, user2, and user3. If we are at Node1, then user1 is the local user and that socket is connected to the user interface process. The local user socket is a UNIX domain socket as opposed to the Internet domain sockets for the remote nodes. The extra complexity of the Internet sockets is necessary

for remote nodes because the UNIX domain sockets only support communications within a node. The domain of the socket is transparent to all of the LISP functions. The primitives send and recv are LISP functions which issue a series of commands to the C process causing it to receive or transmit data on the specified socket.

The Server C process is started by LISP's built-in *process function. This function starts another process and returns a read and write port to that process. The C process behaves as if it were sending its data to the screen and receiving its data from the keyboard. LISP communicates with this process with the standard read and print functions with an additional argument which is the port of the C process. A read or write function normally uses the standard input/output unless the optional arguments are used. Thus to receive input from the C process a command is sent to it by use of a print statement telling it that a read is desired and the name of the socket is to be read. The C process then prints the first packet in that socket to the 'pseudo-terminal'.

When LISP reads the in-port it gets the packet from that particular socket. If the socket is empty a special message is printed so that it is clear that the socket is empty. The packet is an s-expression, so each LISP read will acquire the entire packet. The Server C process is insensitive to the content of the packet, so the Query Planner and Response Planner must ensure that the packets are single s-expressions. If a packet were not a legal s-expression the Server would be able to send the packet, but when the remote node tried to receive it, there would be problems when LISP tried to read the 'pseudo-terminal'. The exact nature of the problem would depend on how the packet was flawed. The processes at a single node are shown in Figure 2. Note that all LISP-to-C interfaces are *process links and all C-to-C interfaces are sockets.

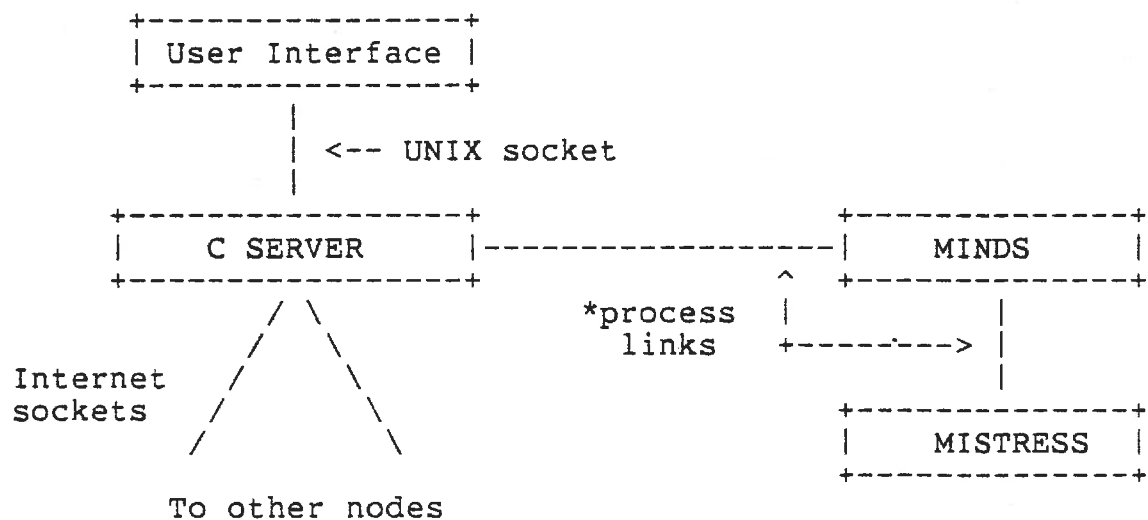


Figure 2. The interprocess links for an individual node.

To receive current inputs the Server reads one of the sockets (the order in which the sockets are polled is unimportant) and examines the first packet. If the socket is empty then a special token is returned and the next socket is examined. If the socket is not empty then the message type (query or response) is determined by looking for specific labels that are unique to either a query or a response. All packets are in the form of association lists. An association list is a list of pairs in which the first element is the key to that field. There are functions in LISP for extracting the pairs from an association list.

The Server identifies a query by the presence of the key 'message'. A response is identified by the key 'result'. These particular keys are not the only means by which queries and responses can be distinguished but they are sufficient. If the packet is a query then it could be from another node or from the local user. The query, whether local or remote, is placed in query_buffer_1. If the packet is a response then the response_blackboard must be updated. Responses will come only from remote nodes since the user will not issue a response and local responses (the results from Mistress) are placed directly in the response_buffer by the Query Planner.

To update the response_blackboard, the slot for the matching query id must be found. The incoming packet will contain both results and metaknowledge although the value of either may be nil. These two fields and name of the originator of the response

are stripped from the packet and the rest of the packet is discarded. The results and metaknowledge are appended to the slot and the name of the originator is removed from the list of users to respond for that slot. The updated slot is then placed on the response_blackboard in place of the old response. The reason for updating the list of users to respond is so that the Response Planner can determine when all the outstanding queries have been answered. The Server continues to check the sockets until all three sockets have been examined and have been emptied.

The other half of the Server's job is to transmit completed responses and remote queries. The Query Planner places all queries which should be transmitted into a special buffer called 'to_remote'. The Server checks the to_remote buffer and transmits each packet until the buffer is empty. Each packet contains a field called destination. The value of that field is used as the argument to the send function. The Server next checks the response_blackboard to see if any of the responses are complete. The Response Planner will mark any completed response by inserting a field labeled 'status' with a value of 'complete'. This indicates that the response is ready to be transmitted. The Server then sends the response to the query originator, which could be a local user or a remote node.

The Server cleans the response_blackboard blackboard and deletes the pointer to the control block for that query. The Query Planner creates a control block to keep track of the status of each query. A unique query id is generated for each query and this query id is the key to the control block. The query id is unique for the entire network. If the key to the control block is not deleted then memory space would be taken up by the control blocks for queries which have been answered.

The control structure for the MINDS program consists of a single loop. Each major module, the Query Planner, the Response Planner, and the Network Server, is called in sequence during each pass through this loop. Most queries require multiple passes to be completed.

The modules work on a set of queries as information becomes available. If part of a response is due from a remote node then the Response Planner awaits that remote response in order to complete the slot. There can be a considerable number of steps involved in a content based query where multistep planning is required. Due to this step-wise nature of answering queries, a given node may contain many queries and responses in various stages of completion. The data structures and the communication facilities are designed to deal with all possible cases of query processing in a distributed environment. The architecture for the entire MINDS system is shown in Figure 3.

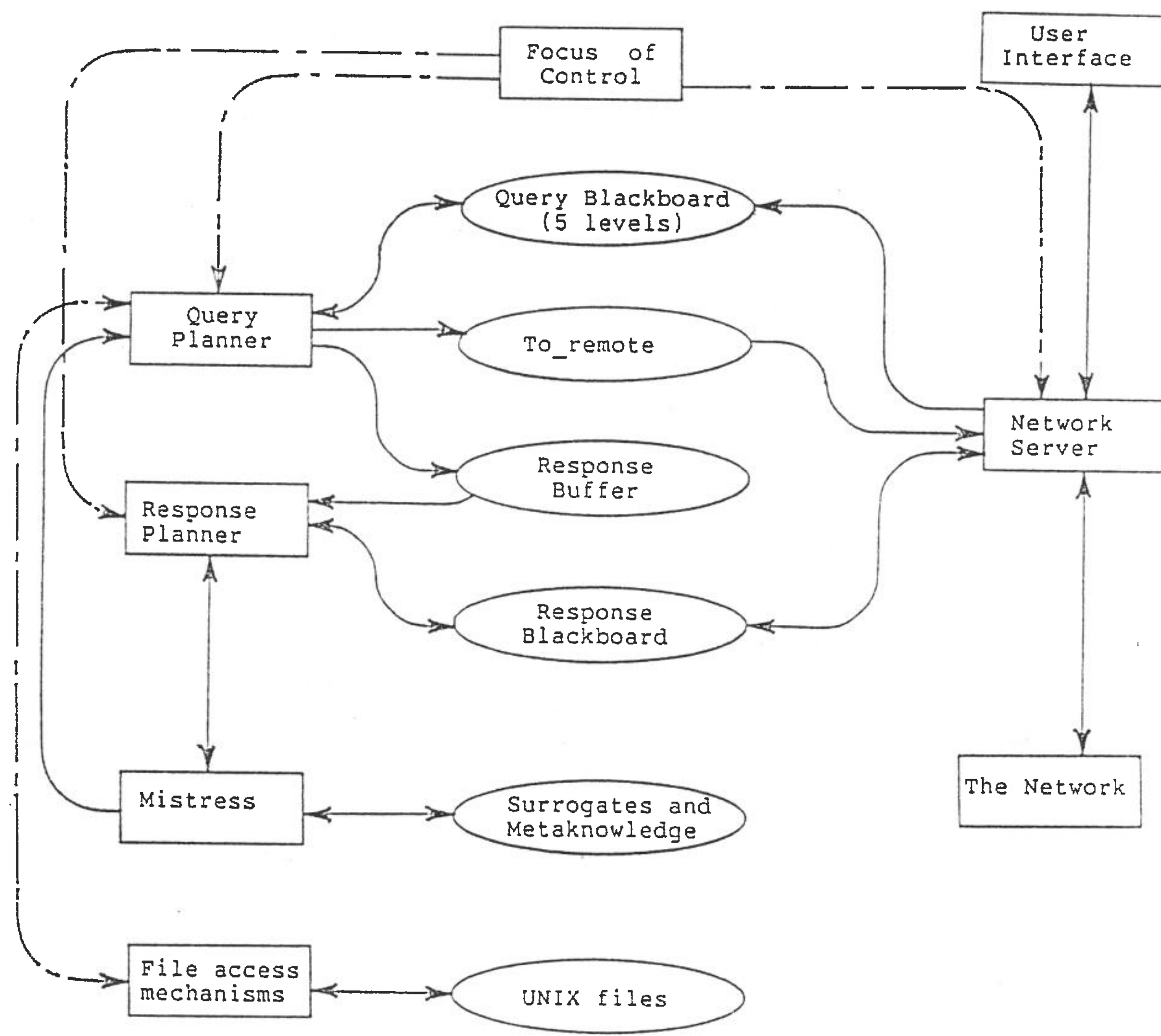


Figure 3. The Architecture of a Single node.

Note: The learning module is embedded in the Response Planner.

— - — - — — A Control Path

———————— A Data Path

B. The C Implementation

This section describes the C code portion which creates the sockets for the network. As previously mentioned, the C Server is a process which creates the sockets and connects with the other C processes at the other nodes. After the sockets have been successfully initialized, the process remains in a 'forever' loop to service requests for network communication. The program is designed to take input from the keyboard and send its output to the terminal. This is done so that the *process function in LISP would work correctly.

The C process reads and writes to the 'pseudo-terminal' set up by the *process function. An alternative to using the *process function was to use LISP's 'cfasl' function whereby the executable C code would be loaded into the LISP environment as a foreign function. There is no particular advantage to either approach in terms of complexity so it was felt that having a separate process would be an appropriate and logical partitioning. Having a separate process also allows for the use of the Server to be more easily extended to use in other LISP programs where such communications are required. Since the Server is a stand-alone process it can also be used directly by other C programs, either by being incorporated in them or by being run concurrently.

Due to the topology of the network each node requires a communication channel to the other nodes as well as to the user interface. There are three sockets at each node. Two are connected to the other nodes and one is connected to the user interface. The sockets which are connected to the other nodes are Internet-domain sockets and the socket for the user interface is a UNIX-domain socket. The UNIX-domain socket is simpler to set up but cannot be used for intermachine communications. Therefore, the Internet sockets must be used for node-to-node communications. The socket type in all three cases is the 'stream' type. The 'datagram' type was ruled out because it is not guaranteed to be reliable or unduplicated. The other type of socket, the 'raw' socket, is not supported and exists primarily for those who wish to experiment with new protocols.

The server-client paradigm causes some awkwardness in setting up a fully-connected network. To help avoid confusion when the term server is used in the context of the server-client relationship, we will write server in lower case; the Network Server will always be capitalized. For a communication channel to be established two processes must play asymmetric roles in the establishment. The server must be active first and await connection from the client. This asymmetry means that the three programs must be set up to run in a specific order. If a client runs before the server then there is nothing for the client to connect to and the connection fails. In the MINDS system, the three C programs must be run in the following order: Node1, Node2, Node3. Once the communication channels are established, full duplex data transmission may occur with no restrictions on

order.

The first program, Node1, acts as a server to Node2 and Node3. Node2 acts as a client to Node1 and a server to Node3. Node3 acts as a client of both Node1 and Node2. After the three nodes have set up their intermachine sockets, each acts as a server awaiting a connection from the user interface process. This last connection is the UNIX socket. Since the accept call which is issued by the server is a blocking call, all sockets must connect successfully before any may begin communicating.

Table 1. The server-client relationship for the socket creation.

Step	Node1	Node2	Node3
1.	server <------->	client	
2.		server <------------>	client
3.	server <------------------------------	--	> client
4.	server (of user interface)	server (of user interface)	server (of user interface)

Note: The user interface socket at each node can be started in any order.

Address binding in the Internet domain is somewhat complicated and requires an explanation. The two processes which are connected through an Internet socket are bound by an association. This association is a five-tuple made up of the following: 1) the protocol, 2) the local address, 3) the local port, 4) the foreign address, and 5) the foreign port. Two C functions are required to fill the association: 'gethostbyname' and 'getservbyname' (only a client needs these functions). The gethostbyname requires the logical host name of the server. The bind call specifies half of the association and the connect call completes the other half. The logical host name is contained in the file /etc/hosts along with the Internet address. For example, the host name may be 'cmil' and the Internet address may be 192.9.200.1. This address is fixed by the manufacturer and is contained on a PROM on the processor board.

This host name is not the same as the address mentioned above as part of the association. A wildcard is used in place of the local address in our implementation allowing the system to

choose the local address. The port may also be left as a wildcard, but the port and local address cannot both be wildcards. The logical host name is supplied as an argument to the C process by LISP's *process function. Thus to change to a different set of machines the arguments in the LISP code which activate the C processes must be changed. The function getservbyname looks in the /etc/services file for a port corresponding to the name and protocol given it. The port can be specified in the C code as a constant but it is better to enter the desired port in /etc/services. We are presently using the name 'mindsport' with a value of 1151. The port number must be higher than 1024. All ports lower than 1024 are privileged and can be accessed only by the system or the superuser. These privileged ports support features such as remote logins.

The packets used for transmission by the sockets are in a C structure. This structure is created by the Server that sent the packet, representing yet another layer of protocol. The structure contains two fields: 1) the length of the packet in bytes, and 2) the contents of the message. The first field is four bytes and contains a long integer representing the length of the message in bytes. The second field is a character array which can have a maximum size of 4 kbytes. The Server first uses a recv call to preview the packet and extract its length. The recv call (this is the recv call in C, not the one we wrote in LISP) allows one to look at the data in the socket: the data is treated as unread and a subsequent read can capture this data. If there is a message in the socket then a read is performed. A non-blocking option allows us to tell if the socket is empty after the first recv. The read uses the length of the message plus four (four bytes for the length) to extract the entire packet. This method in which each packet contains its own length allows packet boundaries to be preserved in the socket queues. It needs only one read call after the length is found. The packet must always be a LISP s-expression. After the packet is read it is printed to the pseudo-terminal. A LISP read is performed from the pseudo-terminal and passed into the MINDS program. A LISP read always reads one s-expression. This is very handy since the packets are s-expressions: each LISP read will acquire one packet and there is no need for extra code in LISP to determine packet boundaries.

One key feature of the sockets is that they are all non-blocking for reads. Normally when a read is performed, the read or recv function will not return until there is something there to read. This is unacceptable since the reading of empty sockets would halt the system and a deadly embrace is likely. The C function 'fcntl' is used to set the proper bit mask so the the sockets become non-blocking. If an attempt is made to read an empty socket, the read returns prematurely and the C global variable 'errno' is set to 35 which is the code for a blocking read condition. To take advantage of this feature the Server resets the global variable 'errno' at the top of its main loop. It performs a recv on a particular socket to get the length of the packet. Before reading the message, if any, it checks the

value of 'errno'. If 'errno' has been changed to indicate a blocking read would occur (to a value of 35), then the Server knows that the socket was empty and returns a special token indicating an empty socket. If the value of 'errno' has not been changed then the packet is read and is passed to the LISP program.

There is a simple command set to tell the C process which socket should be read or written to. For instance, if we wish to send a message to Node2 we would first send the command 'w2', for write user2, to the C process. Then when we type in the packet it is sent to the requested destination. Note that we are using a pseudo-terminal and our packet is delimited by a carriage return. Both the command and the packet are read using the 'get' function. If we want to read from Node2 we send the command 'r2' and the C process prints the first packet on that socket queue to the terminal.

When the time comes to discard the sockets, the LISP function 'stop_soc' should be used. This function closes the Internet sockets and shuts down the UNIX socket. If the program is exited abnormally (a crash for instance) then there will be a process which holds the addresses that will be needed for a MINDS reboot. These processes must be killed prior to rebooting.

III. QUERY PLANNER

A. Structure of the Query Blackboard

The queries from the local Natural Language Interface, remote nodes, the Response Planner are queued in the Query Blackboard. There are five query queues on the blackboard, namely, query_buffer_1, query_buffer_2, query_buffer_3, query_buffer_4 and query_buffer_5. Query_buffer_1 stores the initial queries either from local or from remote nodes. The queries in this queue are in associate list structure. Query_buffer_2 stores a list of new users to be accessed for multistep planning. The queries in this queue are composed of a query ID and a list of users. This queue is updated by the Response Planner.

Query_buffer_3 stores the pathnames for remote copy queries, which are responses to a slave query of "master" copy queries. For example, "Copy all the documents where keyword = unix." This queue is updated by the Response Planner and consists of a query ID and a list of pathnames. Query_buffer_4 stores two sets of metaknowledge table records. The first set is the old records before they are updated; the second set is the revised records. The old ones are deleted from the database and the new ones are inserted. Query_buffer_5 stores surrogate records to be updated.

B. Query Planning

An initial query is parsed by the query parser, and a control table is set up for this query. If the query requires single step planning for a single destination, it will be passed to procedure1 for execution. If it is requires single step planning for multiple destinations, it will be passed to procedure2 for execution. If it requires multistep planning for multiple destinations, it will be passed to procedure3 for execution. If the query is a multistep planning copy query, it will be passed to procedure9 fro execution.

The queries to the local database will be queued in the to_local buffer after planning, which will be processed by the database interface. The queries to the remote nodes will be queued in the to_remote buffer through the network server. For a query requiring multistep planning, a new query planning cycle is started whenever a new list of users to access is appended to query_buffer_2. For a copy query, the remote copy queries are sent out when a list of pathname is ready.

C. The Database Interface

The database access is a lisp interface to the MISTRESS DBMS system. When initialized for a select query, it first issues the query to MISTRESS, then gets the response from the port. Next it

places the result in an associate list structure and queues it in the response_buffer for response planning. For other queries, the database interface simply issues the commands directly to MISTRESS via the lisp port.

c85; D. Query Structure

Since the Query-Planner is coded in lisp, it can accept only an input query expressed as a lisp structure. The use of the following syntax ensures that commands received by the query planner are in the proper format.

```
<query> ::=  '(' '(' 'query_id'     <query ID>     ')'
                 '(' 'originator'   <originator>   ')'
                 '(' 'destinations' <destinations> ')'
                 '(' 'message'      <message>      ')'        ')';

<query ID> ::= <a_string>;

<originator> ::= <a_string>;

<destinations> ::= '(' <a_string> {<a_string>} ')' | '*' ;

<a_string> ::= <first char><other chars>;

<first char> ::= 'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|
                 'i'|'j'|'k'|'l'|'m'|'n'|'o'|'p'|
                 'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|
                 'y'|'z';

<other chars> ::= {<first char>} | {<numeric char>} | <other chars>;

<numeric char> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';

<message> ::= '(' <MISTRESS_query_message> ')' |
              '(' <copy_file_message> ')';

<MISTRESS_query_message> ::= '(' <command> {<argument>} ')'
                             { '(' <command> {<argument>} ')' };

<command> ::= 'select' | 'from' | 'where' | 'update' |
              'set' | 'insert into' | 'values' | 'create table' |
              'drop table' | 'delete' | 'empty table';

<argument> ::= <table-name> | <attribute> {'comma' <attribute>} |
               <requirement> | <value list>| <empty>;

<copy_file_message> ::= '(' <copy command> <from part> <to part>
                            <where part> ')';

<copy command> ::= '(copy)';

<from part> ::= '(' 'from' <pathname list> ')' |
                '(' 'from' '*' ')'
```

```
<to part>  ::= '(' 'to' <new file name> ')' |
               '(' 'to' '*'  ')';

<where part> :: '(' 'where' <requirement> ')';
```

The definition above is made only in terms of lisp structure, not in terms of query syntax. Please refer to the MISTRESS syntax BNF except for the following changes:

- o A ',' should be replaced by 'comma'.
- o A ';' should be omitted.
- o If there are internal parentheses in a MISTRESS query, either 'right-parenthesis' or 'left-parenthesis' should be used instead of '(' or ')'.
- o Single quote ' should be replaced by either "left-quote" or "right-quote".

IV. RESPONSE PLANNER

A. Response Blackboard Structure

The format of the Response Blackboard is as follows:

Response Blackboard (RBB) = ({response-slot})

```
response-slot = ((query_id <id>)
                 (users_to_respond ({<respondent-name>}))
                 (originator <user-name>)
                [(status complete)] Optional:
                       Exists only when status is complete
                 (metaknowledge ({<collapsed-metaknowledge>}))
                 (result ({<synthesized-result>}))
                              )
```

query_id = the tag of the query identification

id = identification of a query, uniquely assigned by the User Interface.

users_to_respond = the tag of a list of user names that have not responded to the query

respondent-name = user-name of the respondent

originator = the tag of the user originated the query

user-name = uniquely assigned by MINDS

status = it is set to "complete" when the response is completely synthesized and ready for the Network Server to transmit. Note: Status exits only when it is set to "complete".

collapsed-metaknowledge = ({metaknowledge grouped by user})

synthesized-result = ({<respondent-name>
<results grouped by respondent>})
Note: The results are grouped in descending order of the certainty factor of each respondent.

B. Global Variables for Response Planning

The response planner uses the following global variables:

1. local_users
   This is the name assigned to the user of the local node and is established when MINDS begins execution.

2. query_buffer_2. This is a list which consists of the names of the users who will be queried based on metaknowledge provided by other respondents. This list informs the Query Planner to perform further planning.

3. query_buffer_4. This is a list which consists of recalculated metaknowledge. The Query Planner uses this list to update the metaknowledge table of the database.

4. response_buffer. This is a list which consists of new queries set up by the Query Planner. The Response Planner retrieves the data from this buffer to set up the response slot on the rbb. After the retrieval, the Response Planner deletes this buffer.

5. rbb. This is the actual Response Blackboard which is initialized when MINDS begins execution. The Response Planner adds response slots to the rbb; the Network Server removes a response slot when the response contained in the slot is selected for transmission.

6. *threshold*. This is a parameter specified by the system. Target users having a certainty factor above this value will be queried.

C. Response Planner Operation

The response planner performs the following functions:

1. Given a query on the response_buffer, the Response Planner creates a corresponding response slot on the rbb. At the same time, if any local metaknowledge and document surrogates are obtained from the query engine, they are placed in the slot. If the response is for a remote query (a query originated from a remote node), the status is set to "complete".

2. Based on the metaknowledge, the response planner creates a list of users to query. A user is selected to be queried if the certainty factor for such an user is higher than the specified threshold. This list of users provides information to update users_to_respond as well as query_buffer_2. Users_to_respond is placed on the response slot in the rbb. Query_buffer_2 is used by the Query Planner to plan the query.

3. The users_to_respond list is used to keep track of the users that have not responded to the query. For each response that the Network Server appends to the rbb, the corresponding respondent is deleted from the users_to_respond list. If users_to_respond is empty, then responses have been returned from all the queried users.

4. For each response slot, the Response Planner determines if all the response are in. If yes, it then synthesizes result and collapses metaknowledge. If no, the slot is skipped.

5. Based on the collapsed metaknowledge, new metaknowledge is calculated. Two layers of heuristic are used to calculate the metaknowledge. These heuristics are discussed in the Learning Subsystem. After the metaknowledge has been recalculated, any user that has a certainty factor higher than the specified threshold is queried.

6. These new users that are to be queried are placed in the the users_to_respond of the response slot and the query_buffer_2.

7. The new metaknowledge is placed in the query_buffer_4 by the Response Planner. Query Planner retrieves the metaknowledge from this buffer and updates the metaknowledge table of the database.

8. When the users_to_respond list is empty and no other users are to be queried, the results are synthesized and the status of the response slot is marked with value "complete". This informs the Network Server that the response is ready to be sent to the originator.

9. After each response slot on the rbb is examined and processed, control is returned to the Focus of Control module.