Remote Predicates For Prolog: A Basis for Declarative Client/Server Applications

by
Kevin Rintoul
B Sc , University Of Victoria, 1988

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

ACCEPTED
FACULTY OF GRADUATE STUDIES
DEAN
DATE 17 May 94

We accept this thesis as conforming
to the required standard

Dr Michael Levy (Department of Computer Science)

Dr Nigel Horspool (Department of Computer Science)

Dr Roger Davidson (Department of Mathematics and Statistics)

Dr Ged McLean (Department of Mechanical Engineering)

© Kevin Rintoul, 1994

University Of Victoria

All rights reserved This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author

Supervisor Dr Michael Levy

ABSTRACT

A number of recent implementations of Prolog contain primitives that allow Prolog to be used to write distributed applications Many of these implementations are concerned with improving the execution times of Prolog programs but most of this work has necessitated a fairly comprehensive extension to Prolog to cope with the issues raised by parallel and distributed execution In contrast, this thesis is concerned with the implementation of simple primitives that enable Prolog to be used to write client-server applications The implementation presented in this thesis is based on the client-server model and consists of three levels At the top level, two Prolog predicates *rcall/2* and *listen_to/1*, are provided that allow for the remote execution of Prolog predicates The semantics of these predicates closely approximate the semantics of Prolog's *call* predicate with the exception that the predicate submitted for remote execution, executes against a remote Prolog database Communication between the client and server is provided by an extension to the standard Edinburgh I/O model Communication between the client and server is hidden from the user by the top level implementation The remote predicates are portable and have been successfully implemented on a number of different computer platforms

Examiners

Dr Michael Levy (Department of Computer Science)

Dr Nigel Horspool (Department of Computer Science)

Dr Roger Davidson (Department of Mathematics and Statistics)

Dr Ged McLean (Department of Mechanical Engineering)

TABLE OF CONTENTS

Page

ABSTRACT ii

TABLE OF CONTENTS iii

LIST OF ILLUSTRATIONS v

Introduction 1

1.1 Related Work 1

1.2 Client-Server Software 2

1.3 Prolog and Client-Server Applications 5

1.4 The Remainder of this Thesis 6

Prolog 8

2.1 Overview of Prolog 8

2.2 A Prolog Example 9

2.3 The Order of Evaluation 12

2.4 Backtracking 13

2.5 The Prolog Library 16

2.6 Prolog and Meta-Programming 17

2.7 Implementing Prolog 20

The Channel Input-Output Model 22

3.1 The Edinburgh I/O Model 22

3.2 Prolog View of Channels 26

3.3 Other Possible Extensions 28

3.4 The Implementation of Channels 29

3.5 Modifications to the Interface Level 29

3.6 Modifications to the Library 32

Prolog Implementation of Remote Predicates 37
4.1 The Semantics of rcall/2 and listen_to/1 37
4.2 The Basic Implementation of Remote Predicates 39
4.3 Dealing With Failure 39
4.3.1 Communicating Failure 40
4.3.2 Implementing Distributed Backtracking 42
4.4 Redirection of user_input and user_output 43
4.5 Prolog Implementation of the Remote Predicates 47
4.6 Technical Issues 57
4.6.1 Marshaling of Data 57
4.6.2 The Server's Ability to Handle Multiple Clients 59
4.6.3 Choice of Programming Language 60
4.6.4 Dealing with Cut Within Client Programs 60
An Example Application Using Remote Predicates 65
Related Work 69
6.1 IC-Prolog II 69
6.2 Delta-Prolog 71
Conclusions 76
APPENDIX A 78
BIBLIOGRAPHY 82

LIST OF ILLUSTRATIONS

Figure | Page

Figure 1 Execution Trace for Predecessor Function 14

Figure 2 Object-Oriented Extension to Prolog 18

Figure 3 Rectangle and Square Object In Prolog 19

Figure 4 The Edinburgh I/O Model 24

Figure 5 Example showing deficiency of Read and Write 25

Figure 6 write_canonical 25

Figure 7 Channel Extensions to the Edinburgh I/O Model 27

Figure 8 A client-server program using extensions to Edinburgh I/O Model 28

Figure 9 Module Structure of the Channel I/O Subsystem 30

Figure 10 Header file for the Channel data type 31

Figure 11 Implementation of See/1 and Tell/1 33

Figure 12 Character-At-A-Time Library Predicates 34

Figure 13 Channel Redirection 35

Figure 14 Basic Implementation of Remote Predicates 39

Figure 15 Server State Transition Diagram 40

Figure 16 Client State Transition Diagram 41

Figure 17 Prolog Definition of Repeat 43

Figure 18 Basic rcall Enhanced With Repeat 44

Figure 19 go_remote Implementation 45

Figure 20 The Implementation of RCall 57

Figure 21 Prolog Based Gopher Client 67

Figure 22 Prolog Based Gopher Server 67

Figure 23. Client-Server Skeleton in IC-Prolog II 70

Figure 24. Squares Example in Delta-Prolog 72

Figure 25. Remote Predicates in Delta-Prolog 73

Chapter I

Introduction

A number of recent implementations of Prolog contain primitives that allow Prolog to be used to write distributed applications. Many of these implementations are concerned with improving the execution times of Prolog programs by exploiting the inherent parallelism of Prolog programs [CC93] [GR87] [KW92] [SH83] but most of this work has been ambitious and has necessitated a fairly comprehensive extension to Prolog to cope with the issues raised by parallel and distributed execution. While these efforts are credible, this thesis is concerned with the implementation of simple primitives that extend the class of programs that Prolog can be used to write, namely client-server applications.

1.1 Related Work

There are two trends in writing network aware applications. The first is to write a distributed application and the second is to write an application for a distributed data-processing system.

A distributed application is an application consisting of two or more programs usually running on different computers, communicating with each other to solve problems [CC93]. The goal of a distributed application is not to increase performance or to increase fault tolerance but to solve problems that would be impossible without communication between programs.

On the other hand, a distributed data processing system is a system with the following attributes: [EN78]

1. Multiplicity of general purpose components. There should be a pool of similar components that can be assigned to specific tasks dynamically.

2. Physical distribution of these components. These components should be connected to a network and should use a protocol to control the transfer of information.

3. A high level distributed operating system. The high level operating system coordinates each of the distributed components. Each component may have its own local operating system.

4. Service usage by name and not by location.

5. Cooperative Autonomy. Each component should be able to refuse to provide service should a more pressing request be outstanding.

The goal of a distributed data processing system is to increase program performance and reliability. Many researchers have noticed the tremendous amounts of idle time in network connected workstations and have sought to exploit it by creating systems that would allow applications to run components in parallel among these workstations.

1.2 Client-Server Software

There is a noticeable trend away from centralized main-frame computers in favor of personal workstations connected to networks. This is in part due to the decrease in price of workstations and the increased availability of networking services. The price of computers has dropped dramatically. It is possible to purchase a computer that is twice as powerful and half the cost of one purchased only a few years ago. It is also possible to purchase user-installable network starter kits through the mail. Direct connections to Wide Area Network are also

becoming commonplace. Most new universities, colleges and government buildings are constructed with data communication facilities in place and many existing facilities are having communication facilities added. With this kind of growth, it is not surprising that there is an increase in demand for network aware applications.

Client-server applications [CS93] are a class of application that have seen growth in popularity partly because of the new availability of computer networks. These applications consist of at least one client computer and one server computer communicating across a network. Under the client-server model, the server is started first and then waits passively for clients to connect. When a client connects it wakes up the server and sends it a request. The server satisfies the requests and returns the result to the client. When the session is over, the server goes back to sleep and waits for other clients.

The client-server model is a common organization among distributed applications. Many well-known systems are modeled after the client-server model, most notably, the SUN-NFS file system. The client-server model solves many problems associated with writing distributed applications. For example, a critical problem in distributed applications is known as the rendezvous problem. This problem occurs where two programs that are part of a distributed application are started simultaneously. One attempts to communicate with the other but the other is not ready to communicate and the application fails. The client-server organization solves this problem by dictating that the server must begin first.

The client-server model has other benefits. If properly designed and well implemented, a client-server application will decrease the demand on network resources when compared to other application architectures because of the

limited amount of communication needed between the client and the server. The client sends the server a request and the server responds with an answer. Intermediate results are not communicated.

The client-server model is often used to implement databases cost effectively. As is typical with client-server applications, database queries are sent to a server by a client and the server responds with the results. These queries may be quite complex and involve searches through many database tables. Using the client-server model, the database implementor is able to use a powerful computer as the database server and reasonably inexpensive workstations as clients. The powerful server ensures that results are computed in a timely manner. As the numbers of clients grow and the demands on the server increase, only the server need be upgraded with more memory, bigger disk drives or more processors in order to maintain acceptable performance levels. Other database organizations would require each of the client workstations to be upgraded as well.

Another example of a well-known client-server application is the distributed document delivery system called Gopher. The Gopher system consists of Gopher clients and Gopher servers. A Gopher client contacts a Gopher server and retrieves the list of information that the Gopher server is able to provide. The Gopher client presents this list to the user in a menu. When the user makes a selection, the Gopher client reestablishes contact with the Gopher server, retrieves the corresponding document and presents it to the user.

Many universities and government agencies are using the Gopher application as an inexpensive method of distributing information. It is not uncommon to see a Gopher system used to distribute university course descriptions, job postings, campus directories or hours of operation of campus

facilities The current gopher implementation is text based but extensions to it are ongoing to allow it to be used to distribute computer graphics, program files and other data formats

1.3 Prolog and Client-Server Applications

Prolog is a programming language based on predicate logic In many situations, Prolog programs are more concise than comparable programs written in an imperative language such as C Many tasks that must be programmed explicitly in conventional languages are handled automatically in Prolog Prolog programs sometimes run slower than equivalent programs written in other languages but this performance penalty is often offset by the other benefits gained by using Prolog such as increased programmer productivity

Standard Prolog is not suitable for writing distributed applications because it lacks the ability to communicate Communication is essential if programs are to exchange information, synchronize and cooperate to solve problems It has become apparent that the class of programs that Prolog could be used to write would be extended if Prolog were able to communicate with other programs

This thesis describes a simple extension to Prolog that allows two Prolog programs to communicate with each other To be consistent with the high level nature of Prolog, this communication mechanism was designed to hide low-level details of inter-process communication from the programmer and required very little modification to Prolog's standard operational semantics

The implementation is based on the client-server model and consists of three levels At the top level, the extension is achieved by two predicates named *rcall* and *listen_to* The *listen_to* predicate makes its Prolog database of facts and predicates available to remote Prolog programs The *rcall* predicate requests the

execution of a query on a remote system These two predicates communicate using lower level communication extensions based on an enhanced I/O model that allows network communication and message passing

The *rcall* predicate has simple semantics based on Prolog's *call* predicate with the exception that the predicate executes against a remote database The concept of the predicate is central to Prolog and therefore all Prolog programmers are familiar with it This simplicity is deliberate Very few new concepts need be learned, enabling programmers to begin using the extensions immediately Adopting this simplicity sacrifices very little power The number of applications that can be written using these extensions is the same as the number of applications that could be written using other communication extensions

The remaining questions this thesis answers is how should *rcall* and *listen_to* be implemented? and how should this implementation deal with the number of interactions between the client and server caused by the use of these predicates? As a simple example, consider a remotely executing query that generates output to the standard output device Clearly, this output should appear locally as output on the computer that raised the query, not remotely This suggests that any Prolog I/O model designed on the assumption of a single non-networked machine must be redesigned to account for the possibility of remote redirection

1.4 The Remainder of this Thesis

Prolog is a language well suited to problems involving objects and relationships between objects Prolog also has a number of features that make it

suitable for writing meta-programs (programs that take other programs as input). For this reason, chapter two discusses Prolog in more detail.

These remote predicates were implemented as extensions to an on-going research Project on multi-paradigm programming called TOPIC - Translator of Prolog Into C [LH93]. The Prolog portion of the topic system consists of three levels: the low level WAM based inference engine[HA91], an interface level and a library level. The inference engine is used by the interface level. The interface level includes the routines necessary to build a basic Prolog system. The library contains the standard built-in primitives common to Prolog systems. Chapter three discusses an I/O model common to many Prolog systems called Edinburgh I/O Model and describes the modifications that were made to the interface level to allow this I/O model to be used for network communications.

The high level remote predicates *rcall* and *listen_to* were implemented as library routines and therefore may be found in the library level of the TOPIC system. Chapter four deals with the implementation of these primitives.

To demonstrate the effectiveness of these extensions, a Gopher application was implemented. The proposed minimal extensions enabled the writing of a very concise gopher-server and a reasonably concise gopher-client. Chapter five describes the implementation of this application and compares it to a similar one, implemented in C.

Chapter 2

Prolog

This chapter provides a brief overview of Prolog. Following the overview, the characteristics of Prolog that affect the remote predicate implementation are discussed in more detail.

2.1 Overview of Prolog

Logic-based programming languages were developed in recognition of the power of formal logic, both in expressing problems, and in describing how these problems may be solved. Logic based programming differs fundamentally from conventional programming because the logic programmer describes the logical structure of problems rather than prescribing how the computer is to go about solving them. The logic programmer states facts about objects and relationships between them, and the logic programming language makes inferences based on these facts. Prolog is a well-known example of a logic-based programming language.

Imperative programming languages have often been criticized for being an unnatural medium with which to express problems and solutions to problems. Imperative programming languages are based on an abstraction of the von Neuman architecture. The von Neuman architecture is characterized by a large uniform set of memory cells, a processing unit and some local cells called registers. The processor possesses instructions for fetching items from memory, manipulating these items and writing them back again. This architecture is possible to implement efficiently in hardware, but is not an ideal model to use when expressing and solving problems. Many feel that languages based on this

model, such as C or Pascal, have caused a bottleneck for programmers using them to the extent that their productivity is impaired [BA78] Experience has shown that some problems are much easier solved using logic-based programming languages such as Prolog than by using the more traditional imperative languages such as C or Pascal Using logic-based languages, it is often possible to express the underlying problem and hence the solution to the problem more directly In these cases, the programmer can be more confident of the correctness of the solution

Prolog is a logic-based language centered around a small set of basic primitives including tree-based data structures, pattern matching and backtracking These basic primitives relieve the programmer of many common mundane tasks inherent in imperative languages such as memory management, and data structure manipulation

A Prolog program consists of facts and rules These facts and rules form a set of axioms To use the Prolog system, the programmer asks a question of the system that can be viewed as a conjectured theorem Prolog then tries to prove this theorem by demonstrating that the theorem can be logically derived from the set of axioms An example of this will be shown in the following section

2.2 A Prolog Example

An example of a Prolog program is one that models family relationships

```
father(john,jim).
father(jim,henry).
father(matt,zak).
mother(mary,jim).
mother(jenny,zak).
parentsOf(Father,Mother,Child) :-
      father(Father,Child),mother(Mother,Child).
```

This program consists of five facts and a rule. The facts state who is the father and mother of whom. For example, the first fact, *father(john,jim)*, states that *john* is *jim's* father. Facts are the simplest statements in Prolog. Individuals such as *john* and *jim* are called atoms.

The rule,

```
parentsOf(Father,Mother,Child) :-
      father(Father,Child),mother(Mother,Child)
```

states that two people are the parents of a child if a father has the same child as a mother. Variables in Prolog are denoted by upper case words. The statement to the left of the :- symbol, *parentsOf(Father,Mother,Child)* is called the head of the rule. The statements to the right of it, *father(Father,Child),mother(Mother,Child)* is called the body.

To use the Prolog system, a Prolog programmer asks questions about facts and rules in the Prolog database. For example, to find out who *jim's* parents are in the above example, a Prolog programmer would ask the following:

```
?- parentsOf(Father,Mother,jim).
```

To answer this question, the Prolog system examines the list of facts and rules from top to bottom, in search of a rule head of arity three with the name *parentsOf*. As expected, Prolog finds the rule:

```
parentsOf(Father,Mother,Child) :-
      father(Father,Child),mother(Mother,Child).
```

Because the programmer specified that the child was *jim*, before Prolog proceeds, it first substitutes *jim* for each of the *Child* variables. This substitution process is called binding a value to a variable or instantiating a variable. The predicate may now be rewritten as follows:

```
parentsOf(Father,Mother,jim) :-
      father(Father,jim),mother(Mother,jim).
```

This rule has two sub-goals, *father(Father,jim)*, and *mother(Mother,jim)* Mathematically, this rule can be read as *father(Father,jim)* and *mother(Mother,jim)* implies *parentsOf(Father,Mother,jim)*

To prove the *parentsOf(Father,Mother,jim)* predicate, Prolog first proves each of *parentsOf*'s sub-goals To solve the first sub-goal Prolog finds a match for *father(Father,jim)* in the first fact, *father(john,jim)* and *john* is then bound to *Father* Having satisfied the first sub-goal, Prolog attempts to satisfy the next sub-goal *mother(Mother,jim)* It finds a match in the fourth fact, *mother(mary,jim)* and *mary* is bound to *Mother* The two sub-goals are now satisfied, therefore the original goal is satisfied and Prolog announces its success by stating

```
Father = john
Mother = mary
```

Other questions may be asked of the Prolog system including "who are John and Mary's children?" For example

```
parentsOf(john,mary,Child).
```

and Prolog would reply

```
Child=jim
```

This example, while simplistic, demonstrates a number of interesting characteristics of Prolog One such characteristic is that to create a Prolog program, the programmer simply states the facts that are known and relationships between them No thought is given to how Prolog is to go about solving the problem or satisfying the query Another characteristic is that the programmer needn't be concerned with allocating memory or manipulating data structures Relationships between objects are expressed naturally and directly using Prolog's built-in hierarchical data structures Additionally, little work needs to be performed in order to have Prolog answer a completely different

questions, such as who is John and Mary's child. This is not a claim that many other programming languages can make.

2.3 The Order of Evaluation

One characteristic of Prolog that was glossed over in the preceding example was the order in which Prolog evaluates predicates. This ordering, while having little impact on the logical meaning of a Prolog program, has an effect on a program's efficiency and may even affect whether or not a Prolog program terminates. Prolog, by default, scans for predicates from the top of the program down. For example, consider the following set Prolog facts.

```
dark(brown).
dark(green).
dark(red).
```

If the question

```
?- dark(X)
```

were asked of the Prolog system, it would respond

```
X=brown
```

Subsequent requests for more answers would result in

```
X=green and X=red.
```

When evaluating the bodies of predicates, Prolog performs these evaluations from left to right. For example, when the rule

```
parentsOf(Father,Mother,Child) :-
      father(Father,Child),mother(Mother,Child).
```

is evaluated, *father(Father,Child)* is evaluated first, followed by *mother(Mother,Child)*.

To see that evaluation order is important, consider the following Prolog program that extends the parental relationship program of the previous section to include a rule regarding predecessors

```
parent(Parent,Child) :-
      mother(Parent,Child).

parent(Parent,Child) :-
      father(Parent,Child).

predecessor(Parent,Child) :-
      parent(Parent,Child)

predecessor(Predecessor,Successor) :-
      parent(Predecessor,Child),
      predecessor(Child,Successor)
```

This program states that a person is a predecessor of another person if the first person is the second person's parent, or if the first person has a child that is a predecessor of the second person

Without changing the logical meaning of the program, the predecessor relation may be recast as follows

```
predecessor(Predecessor,Successor) :-
      predecessor(Predecessor,Child)
      parent(Child,Successor).

predecessor(Parent,Child) :-
      parent(Parent,Child).
```

Unfortunately, because of the order in which Prolog evaluates goals the program will never terminate The first sub-goal of the first clause of the predecessor relation will be evaluated infinitely Such situations should be avoided when constructing Prolog programs

2.4 Backtracking

An important characteristic of Prolog is its ability to backtrack When Prolog fails to prove a goal in a given clause, it backtracks to a previous clause

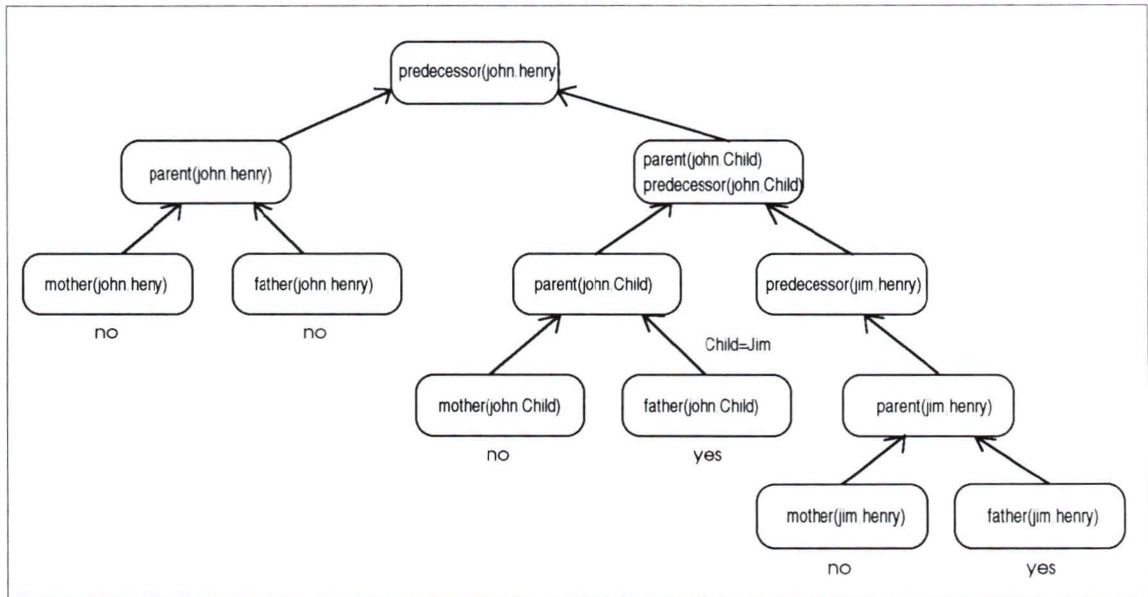


Figure 1 Execution Trace for Predecessor Function

and attempts alternate execution paths in search for a proof As a simple example, consider the parent relation from the preceding example If we were to ask Prolog

```
?- parent(john,jim)
```

Prolog would find two clauses for the parent relation *parent(Parent,Child)* - *mother(Parent,Child)* and *parent(Parent,Child)* - *father(Parent,Child)* Prolog attempts to prove the goal by executing the first goal and fails because there is no fact that states *mother(john,jim)* Prolog then backtracks and executes the second clause of the parent relation and this time succeeds immediately because *father(john,jim)* is in the database as a fact

A more interesting example is the execution of the query

```
?- predecessor(john,henry).
```

Figure 1 shows the execution trace Prolog scans the database from the top down and finds the first predecessor clause In Prolog's attempt to satisfy the predecessor goal, Prolog will satisfy each of the sub-goals In this case, there is one *parent(Predecessor,Successor)* *john* is bound to *Predecessor* and *henry* bound to

Successor. Prolog then re-scans the database in search of *parent(john,henry)*. It finds *parent(Parent,Child)* - *mother(Parent,Child)*. It binds *john* to *Parent* and *henry* to *Child* and then re-scans the database in search of *mother(john,henry)*. Prolog doesn't find a match so it backtracks and begins executing the second parent clause - *parent(Parent,Child)* - *father(Parent,Child)*. *john* is bound to *Parent* and henry is bound to *Child*. Prolog then re-scans the database in search of *father(john,henry)* and fails.

At this point, having no more parent clauses to try, Prolog backtracks to the predecessor function and begins executing the second predecessor clause. This clause has two sub-goals, *parent(Predecessor,Child)* and *predecessor(Child,Successor)*. *john* is bound to *Predecessor* and *henry* is bound to *Successor*. If we care to follow the execution through, we see that *parent(john,Child)* is satisfied by the fact *father(john,jim)*. Having satisfied the first sub-goal, *jim* is bound to *Child* and the second sub-goal is executed - *predecessor(jim,henry)*. This sub-goal is satisfied by the fact *father(jim,henry)*.

The mechanism Prolog uses to keep track of alternate paths of execution is called the choice point. Among other things, Prolog maintains a pointer to the next clause to execute in the choice point should evaluation of the current clause fail. For example, executing parent predicate of the previous section for the first time will cause Prolog create a choice point with the next clause pointer set to parent's second clause. When evaluation fails, Prolog examines this choice point and begins executing this clause. If this evaluation fails, there are no more alternate clauses to execute and therefore Prolog discards the choice point and examines a choice point of the most recently executed predicate prior to execution of the parent predicate.

Chapter four demonstrates the special attention that backtracking requires in a distributed environment

2.5 The Prolog Library

Prolog implementations usually come with an extensive set of extra-logical functions Such libraries serve the same purpose as library functions in other programming languages such as C and Pascal They relieve the programmer of having to implement primitive functions that are often needed in everyday use The library is usually extensive and usually includes functions for I/O, arithmetic, performing comparisons, control constructs, meta-programming, manipulating the Prolog database, exception handling, and more A suggested set of predicates for the Prolog library is found in the ISO Draft Prolog Standard

In reality, many of these functions are implemented using Prolog, but a few are not For example, pure Prolog does not include primitives to access an operating system's underlying file system In this case, it is necessary to implement the I/O functions in a lower level language and interface them to the Prolog system

The library is important to the Prolog system implementor because it gives the implementor a method of adding functionality to a Prolog system without changing the meaning of Prolog programs For example, if an implementor would like to add a new I/O model to a Prolog system, he or she may do so simply and directly by adding a new set of Prolog library predicates The underlying Prolog language need not be changed The significance of this will be shown in chapter Four

2.6 Prolog and Meta-Programming

A meta program is a program that manipulates other programs. A very simple example of a meta-program is a pretty printer. Pretty printers read program source code and format it according to a set of rules. Using a pretty printer, a programmer can ensure that his or her program conforms to a standard for indenting, spacing and placement of parenthesis. Other examples of Meta-programs include source level debuggers and program interpreters.

Prolog contains a number of built-in functions for meta-programming. At the heart of Prolog's meta-programming ability is the built-in predicate "*call*". The meaning of *call*(T) is the same as T, i.e. as if T appeared textually in *call*(T)'s place. Call is often used to execute a predicate that is passed to another predicate as an argument. The following Prolog program illustrates the use of *call*.

```
execute(T,S) :-
      call(T),
      S=success.

execute(T,S) :-
      S=failure.
```

This program will execute the predicate T and will set S to success if T executes successfully, otherwise, it will set S to failure.

Consider a more interesting example of meta-programming. Suppose that we would like to perform object-oriented programming using Prolog. Object-oriented programs are constructed from objects and computation results from objects sending messages to one another. When an object receives a message, it executes a procedure corresponding to the message called a method. For

```
send(Object, Message) :-
     get_methods(Object, Methods),
     process(Message, Methods).

get_methods(Object, Methods) :-
     object(Object, Methods).

get_methods(Object, Methods) :-
     isa(Object, SuperObject),
     get_methods(SuperObject, Methods).

process(Message, [Message | _]).

process(Message, [(Message :- Body) | _]) :-
     call(Body).

process(Message, [_ | Methods]) :-
     process(Message, Methods).
```

Figure 2 Object-Oriented Extension to Prolog[1]

example, a rectangle object executes methods that provide responses to the messages *area*, and *draw* The area method might compute the area occupied by the rectangle in square centimeters while the draw method might draw the rectangle on a pen-plotter

A distinguishing feature of the object-oriented paradigm is the ability to create an object from an existing object The existing object is called the parent The new object inherits behavior of its parent in addition to adding new

behavior of its own This feature facilitates code re-use because when a new object is created, the programmer need not implement methods already implemented in the parent, unless a different response to a message is required

[1]From [BR90] pp 548

```
object(shape(X,Y),
       [(location(X1,Y1) :- X1 = X, Y1 = Y),
        (describe :- write('Shape at location '),
                     write(X),
                     write(','),
                     write(Y))]).

object(rectangle(dimension(Length,Width),coord(X,Y)),
       [(area(A) :- A is Length * Width),
        (describe :- write('Rectangle at location '),
                     write(X),
                     write(','),
                     write(Y),
                     send(rectangle(dimension(Length,Width),
                          coord(X,Y)),area(A)),
                     write(' of area '),
                     write(A))]).

object(square(Side,coord(X,Y)),
       [(describe :- write('square at location '),
                     write(X),
                     write(','),
                     write(Y),
                     send(rectangle(dimension(Side,Side),
                          coord(X,Y)),area(A)),
                     write('of area '),
                     write(A))]).

isa(rectangle(dimension(Length,Width),coord(X,Y)),
     shape(X,Y)).

isa(square(Side, coord(X,Y)),
     rectangle(dimension(Side, Side),coord(X,Y))).
```

Figure 3 Rectangle and Square Object In Prolog

As an example of object-oriented programming, consider a graphics system containing a number of *shape* objects The *shape* object might respond to messages such as *location(X,Y)*, and *describe* The response to the *location* message is Cartesian coordinate of the *shape* The response to the *describe* message is the type and location of the shape From the *shape* object, the programmer might create more sophisticated shapes such as *circles* and *rectangles* For these new objects, the *describe* methods will be re-written to report the new shape's type but the

location method may be used, unchanged. Further, a new method called *area* may be added to each of these new shapes that would report the area occupied by the shape.

Native Prolog does not support object oriented programming but by using Prolog's support for meta-programming, adding simple object oriented extensions is straight-forward. Figures 2 and 3 show one possibility for such an extension.

An example using these object-oriented extensions is shown in Figure 3. In this example, to send a *rectangle* a *describe* message, the Prolog user would execute the predicate:

```
?- send(rectangle(dimension(3,5),coord(1,1)),describe).
```

This same example also demonstrates the use of inheritance. Notice the *isa* relation where *square* object inherits the behavior of a *rectangle* object and the *rectangle* object inherits the behavior of the *shape* object.

The built-in Prolog meta-programming operations, as we will soon see, are an important capability that is used extensively in the implementation of remote predicates.

2.7 Implementing Prolog

An argument against the use of Prolog has been that Prolog is run-time inefficient. Programs written in Prolog executed slower than equivalent programs written in imperative languages. While this may have been true in the past, this is not clearly so now.

Many advances in Prolog execution have been made since Prolog was first invented. Most current implementations of Prolog are based on Warren's Abstract Machine [HA91]. Warren's Abstract Machine (WAM) consists of a

memory, instructions and optimizations tailored to executing Prolog programs efficiently. All WAM instructions can be and have been implemented in high level languages and are therefore portable to a wide variety of computer architectures. A Prolog system can be implemented by a program that translates Prolog programs into a set of WAM instructions. These instructions can then be interpreted by a byte-code compiler, or translated themselves into native machine code. Typically, a byte-code compiler is 6-10 times faster than an interpreted program while a native code program is another 4-6 times faster.

The implementation of remote predicates in this thesis is achieved by modifying the TOPIC system discussed in chapter 1. This system was written either in ANSI-C or Prolog translated to ANSI-C and therefore is portable to any environment supporting an ANSI-C compiler.

As one might expect, the primitives that form the top level interface to the remote predicates, *listen_to* and *rcall* were implemented as library functions. The implementations of these functions were performed mostly in Prolog although a small number of ancillary functions, written in a lower level language, were needed. This Prolog system's I/O subsystem was modified slightly to allow for network communication and the library functions that implemented the standard Edinburgh I/O model were extended to allow for network communication. The modification of this system's I/O primitives and the extension to the Edinburgh I/O model is the topic of the next chapter.

Chapter 3

The Channel Input-Output Model

Chapter one pointed out that the ability to communicate is a prerequisite to writing distributed applications It also pointed out that standard Prolog lacks this ability and therefore is not suitable for such applications This deficiency is remedied by extending the Edinburgh I/O model, giving it the capability of sending and receiving Prolog terms across a network

This chapter begins by discussing the Edinburgh I/O model and extensions that were made to it It then discusses the lower level implementation of these extensions and special measures that were taken to ensure easy portability between hardware platforms and network environments

3.1 The Edinburgh I/O Model

The Edinburgh I/O model is standard in most implementations of Prolog The model contains nine predicates for performing I/O to and from files The Edinburgh model is simple and somewhat limited, but when enhanced with networking capabilities, provides the basic functionality needed for inter-process communication Figure 4 is a description of the predicates included in this model The '+' symbol prefixing a predicate's argument indicates the argument is an input parameter '-' indicates an argument is an output parameter '?' indicates an argument may either be an input or an output parameter

There are three characteristics of this model that may not be obvious from the above description The first is that at any one time there are only two active files the current input stream and the current output stream Reads are performed with respect to the current input stream and writes are performed

with respect to the current output stream More files may be open but only two of them are active

---

see(+*File*)

File *File* becomes the current input stream *File* may or may not already be opened by the Prolog system If it is not already open, the file is opened and the current input stream is set to this file If it is already open, the file is not re-opened but the current input stream is set to this file

seeing(?*FileName*)

FileName is unified with the name of the current input stream If the current input stream is *user_input*, *FileName* is unified with *user*, otherwise *FileName* is unified with the name of the file that was last made the current input stream by *see/1*

seen

Closes the current input stream, and resets it to *user_input*

tell(+*File*)

File *File* becomes the current output stream *File* may or may not already be opened by the Prolog system If the file is not already open, the file is opened and the current output stream is set to this file If it is already open, the file is not re-opened but the current output stream is set to this file

telling(?*FileName*)

FileName is unified with the name of the current output stream, If it is *user_output*, then *FileName* is unified with user, otherwise, FileName is

unified with the name of the file last made the current output stream by *tell/1*

told

Closes the current output stream, and resets it to *user_output*

read(?Term)

The next term, delimited by a full-stop (i e a followed by either a space or a control character), is read from the current input stream and is unified with *Term* The syntax of the term must agree with current operator declarations If a call to read(Term) causes the end of the current input stream to be reached, *Term* is unified with the term *end_of_file* Further calls to read on the same input stream will cause failure, unless the stream is connected to the terminal

write(?Term)

The term *Term* is written onto the current output stream according to the current operator declarations

nl

A new line is started on the current output stream If the current output stream is the terminal, its buffer is flushed

---

Figure 4 The Edinburgh I/O Model[2]

The second is that this model is only capable of reading valid Prolog terms If a program needs to read data types such as English sentences, other extensions to this model are needed A common approach is to treat an input stream as a

---

[2]Adapted from SICStus Prolog User's Manual, Swedish Institute of Computer Science, 1992

```
tell('write_example'),
write('ATerm.'),
told,
see('write_example'),
read(X),
seen.
```

Figure 5 Example showing deficiency of Read and Write

sequence of characters and perform reads one character at a time This extension is often implemented as an additional library function called *get/1*

The third is that it is not always possible to read terms that were written using *write/1*, using *read/1* To see that this is the case, consider the following Prolog program

This program will respond *X=_227* meaning that *X* is not instantiated The reason for this apparently peculiar behavior is that *ATerm* begins with a capital letter and in Prolog, terms beginning with capital letters are variables Therefore, when Prolog reads *ATerm* from the file, Prolog believes it is reading a Prolog variable The fact that *ATerm* is actually an atom is lost To solve this problem, a new primitive, *write_canonical*, is needed that will write terms in their canonical form, in the case of *ATerm*, quoted The implementation of remote predicates discussed in Chapter 4 makes extensive use of *write_canonical*

write_canonical(?Term)

Write Term on the current output stream in a format such that it may be read by read/1

Figure 6 write_canonical

3.2 Prolog View of Channels

Only minor modifications to the Edinburgh model were required to extend it for use in a networking environment. Although no new primitives were needed, the semantics of the *see/1* and *tell/1* predicates were changed. *see* and *tell* were modified to allow opening of communication channels across a network. The following figure describes the enhanced *see* and *tell*.

---

see(+*Channel*)

File *Channel* becomes the current input stream. *Channel* may be a file name or a port address of the form *port(ComputerName,PortNumber)*. If *Channel* is a file name, the following action is taken: if the file is not already opened, the file is opened and the current input stream is set to this file. If it is already open, the file is not reopened but the current input stream is set to this file.

If *Channel* is a port address, the following action is taken: if an input network connection to this address is already open, the current input stream is set to this stream. If an output network connection to this address is already open, the current input stream is set to this stream and the output connection is unaffected. (Network connections are bi-directional)

If *ComputerName* is *self*, *see* blocks and waits passively for a process to connect on that port number.[3] An example of a port address is

---

[3] When a client connects to the server, the *see* predicate forks a new process to service the client's request if the Prolog system is running on a multi-tasking operating system that supports process forking. The original process waits for further client connections.

port(csr,2001). Port addresses differ from standard file names in that port addresses are Prolog structures with name *port* and has arity two. File names are Prolog atoms.

tell(Channel)

File *Channel* becomes the current output stream. *Channel* may be a file name or a port address of the form *port(ComputerName,PortNumber)*. If *Channel* is a file name, the following action is taken: if the file is not already opened, the file is opened and the current output stream is set to this file. If it is already open, the file is not reopened but the current output stream is set to this file.

If *Channel* is a port address, the following action is taken: if an output network connection to this address is already open, the current output stream is set to this stream, if an input network connection to this address is already established, the current output stream is set to this stream and the input connection is unaffected.

Figure 7. Channel Extensions to the Edinburgh I/O Model

The capabilities of the enhanced *see* and *tell* are a super-set of the capabilities of *see* and *tell* in the original Edinburgh I/O model. Therefore existing programs using the Edinburgh I/O model continue to function, unaffected. An example of the use of these extensions is shown in figure 8. This client-server program implements a simple form of distributed execution. The server is started first when the user types *server* in response to Prolog's "?-" prompt. The *server* predicate then waits passively in the *see/1* predicate for clients to connect. In a separate Prolog session, requests are made when the Prolog user enters the query *client(Port,Term)* where *Port* is the network address of the server

```
server :-
        see(port(self,2001)),
        tell(port(self,2001)),
        read(Term),
        call(Term),
        write_canonical(Term),
        write('.'),
        nl,
        seen,
        told,
        server.


client(Port,Term) :-
        tell(Port),
        see(Port),
        write_canonical(Term),
        write('.'),
        nl,
        read(X),
        Term=X,
        seen,
        told.
```

Figure 8 A client-server program using extensions to Edinburgh I/O Model

and *Term* is the term to be evaluated Upon connection, the server reads the term sent to it by the client, evaluates it and returns the result

3.3 Other Possible Extensions

Other extensions to Prolog that add inter-process communication have been proposed For example, SICStus Prolog uses a model based on streams where a stream is an open file or file-like object such as a communication channel The SICStus model has each I/O function using an additional parameter, called a handle, to identify the open stream to operate on Unlike the Edinburgh model, the SICStus model does not contain the notion of current

input or output streams While the SICStus model is more flexible, it is also incompatible with the Edinburgh model and therefore the existing base of Prolog programs that use the Edinburgh model would need to be modified

3.4 The Implementation of Channels

The Prolog system used in this thesis is called TOPIC TOPIC is implemented in three levels, the WAM Engine, the Interface and a Library The WAM engine level did not need modification and therefore will not be discussed further The I/O subsystem of the Interface level was modified substantially and will be discussed in the following sections

3.5 Modifications to the Interface Level

The first attempt at adding inter-process communication to TOPIC required little modification to the existing I/O subsystem This system used ANSI C library functions for I/O exclusively Adding inter-process communication was a simple matter of adding code to establish a network connection using the UNIX implementation of Berkeley Sockets The Berkeley Socket interface to C is a simple extension of the UNIX file system Once a conversation between two processes is established, the two processes communicate using the same functions that are used to perform standard file I/O This abstraction was very convenient, but the situation became much more complicated when the TOPIC system was ported to the IBM PC and the Macintosh computers The Macintosh and PC interfaces to TCP/IP are quite different from UNIX's Berkeley Sockets interface

Because of the differences in the TCP/IP application programmer's interfaces on different hardware platforms, it became clear that a more powerful

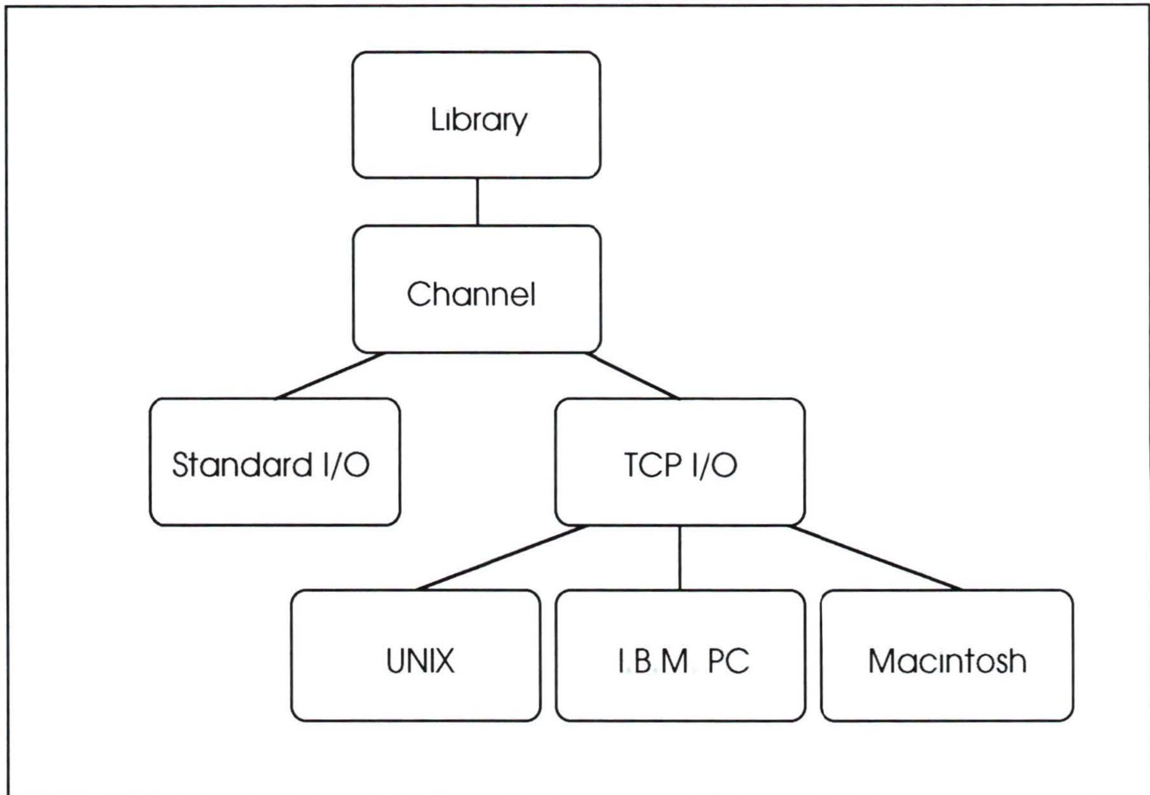


Figure 9. Module Structure of the Channel I/O Subsystem

and more flexible I/O model was needed that would encapsulate these differences and allow easier porting of the TOPIC system to new hardware platforms. To satisfy this need, an abstraction called a *channel* was added that would hide the underlying file and networking system, isolating the platform dependent parts of the I/O subsystem and limiting platform dependent code to less than a few hundred lines of C code. All I/O within the Prolog system would be performed though the Channel interface and the Channel subsystem alone would communicate with the lower level I/O functions. Figure 9 shows the placement of the channel subsystem in relationship with the rest of TOPIC.

It would be unfortunate if the Channel subsystem needed to be modified every time the TOPIC system is ported to a new platform or new network interface. To avoid this, one further abstraction was added: each channel type

```
typedef enum {a_stream,a_port,a_either} channelType;
typedef enum {a_input, a_output } channelDir;
typedef struct _channel{
     int (*read)(int handle);
     void (*write)(int handle, int c);
     void (*unread)(int handle,int c);
     void (*flush)(int handle);
     void (*close)(int handle);
     int handle;
     int prev_col;
    } channel;
extern int AddChannel(String n, channelType t,
                    channelDir d, channel c);
extern int ChannelId(String n, channelType t,
                    channelDir d);
extern void DeleteChannel(int id);
extern void SetCurrentChannel(channelDir d,int id);
extern int CurrentChannel(channelDir d);
Boolean Redirect(int source_id,int target_id);
extern String ChannelName(int id);
extern ChannelType ChannelType(int id);
extern int ChannelFd(int id);
extern void SetLink(int id1,int id2);
extern void Link(int);
extern void SetPrompt(char *);
extern char *GetPrompt(void);

extern void xprintf(const int, char *f,...);
extern void xputc(char c);
extern int xgetc(void);
extern void xungetc(const char c);
extern char *xgets(char *s);
extern void xputs(char *s);
extern void xflush(void);
extern int xline(channelDir d);
extern int xcol(channelDir d);
```

Figure 10. Header file for the Channel data type

was required to implement 5 basic I/O functions, *read, write, unread, flush* and *close*. The interface to these functions is the same for each channel type but their implementations are quite different. For example, the standard file system might implement read and write using the ANSI standard libraries. On the Macintosh, networked reads and writes might be implemented using Mac/TCP. These functions are then assigned to a structure that is used by the Channel subsystem for subsequent calls to these functions.

The advantage of this organization becomes obvious when the TOPIC system is ported to a new platform. For example, to port the system to the Macintosh, a small amount of code is written to implement *read, write, unread, flush* and *close* using the Macintosh's TCP/IP implementation. An initialization function is written to assign these functions to the channel structure and AddChannel is called to link the new communication capability to the Prolog system. Neither the Channel subsystem, nor any other part of the Prolog system need be modified. The use of these function pointers to the basic I/O functions is analogous the concept of virtual functions in an object oriented programming language.

The header file for the Channel subsystem is shown in figure 10. The fully explained interface to this subsystem is shown in Appendix A.

3.6 Modifications to the Library

Following implementation of the channel data structure, modifying the library function *see/1* and *tell/1* with their new networking capability was straight-forward. The following figure shows this implementation.

```
static int see_or_tell(char direction){
String name;
channel c;
channelType ct;
int c_id;
Boolean addOK;

/* WAM fail register */
fail = false;
fail = !streamNameToStr(&name,&ct,direction,
                        A(1),X(2),X(3));

if (fail) return;

c_id = ChannelId(name,ct,direction);
if(c_id == -1) {
          if(ct == a_port)
               addOK = TCPMakeChannel(name,direction,&c);
          else
               addOK = StandardMakeChannel(name,
                              direction,&c);
          if(!addOK) {fail = true; return; };
               c_id = AddChannel(name,ct,direction,c);

     }

     SetCurrentChannel(direction, c_id);
     return;
}
```

Figure 11 Implementation of See/1 and Tell/1

Three functions in figure 11 require explanation *StreamNameToStr*, *TCPMakeChannel* and *StandardMakeChannel* *StreamNameToStr* serves two purposes to determine the type of channel to open and to convert the port name or file name from its WAM representation to a representation that the underlying I/O system is able to use If channel type is *a_stream*, *nameToStr* converts name to a string If the channel type is *a_port*, *nameToStr* converts name to a string of the form *ComputerName PortNumber*

read_ch(?Char)

Read a single character from the current input stream and return it in *Char*. If a call to *read_ch(Char)* causes the end of the current input stream to be reached, *Char* is unified with the term *end_of_file*. Further calls to read_ch on the same input stream will cause failure, unless the stream is connected to the terminal.

write_ch(?Char)

Write the single character *Char* to the current output stream.

unget_ch(+Char)

Pushes the single character *Char* back into the current input stream such that the next call to *read_ch/1* will return *Char*.

Figure 12. Character-At-A-Time Library Predicates

TCPMakeChannel and *StandardMakeChannel* are the initialization functions for the lower level interface to the network system based on TCP/IP and standard file system respectively. Their purpose is twofold: to open the specified channel and to assign the five basic I/O functions, *read, write, unread, flush* and *close* to the channel structure. Following this initialization, channel creation is completed by *AddChannel* and the current channel is set to this new channel.

Only minor modifications to *seen* and *told* were required. For example, where *seen* and *told* used to call the ANSI C function *fclose*, they now call the channel function *DeleteChannel*. Read/1 and Write/1 are unchanged.

Three additional library functions that perform character-at-a-time I/O are required by the remote predicate implementation described in chapter 4. They

are *read_ch/1, write_ch/1* and *unget_ch/1* and their semantics are described in figure 12.

Four predicates form the interface between the I/O redirection capabilities of the channel sub-system and Prolog. Their semantics are described in figure 13. They are specifically used by the server to redirect *user_input* and *user_output* to the communication channel between the client and server.

---

redirect_input(+Channel)

Redirects the current input stream to be from *Channel*. *Channel* must have been previously opened for input. Following redirection, all calls to seeing/1 will report the name of the original, previously unredirected input stream.

redirected_input

Causes the current input stream to be closed and sets the current input stream to be from the stream that was current, prior to redirection.

redirect_output(+Channel)

Redirects the current output stream to be from *Channel*. *Channel* must have been previously opened for output. Following redirection, all calls to telling/1 will report the name of the original, previously unredirected output stream.

redirected_output

Causes the current output stream to be closed and sets the current output stream to be from the output stream that was current prior to redirection.

Figure 13. Channel Redirection

Now that Prolog has the ability to perform inter-process communication, the implementation of remote predicates is possible. This implementation is discussed in chapter 4.

Chapter 4

Prolog Implementation of Remote Predicates

This chapter is concerned with the implementation of the remote predicate library functions *rcall/2* and *listen_to/1*. It begins with a discussion of the semantics of these functions and then proceeds to describe their lower level implementation. The chapter is concluded with a discussion of some of the decisions that were made during their design and some of the technical issues encountered during their implementation.

4.1 The Semantics of rcall/2 and listen_to/1

In a Prolog system, the meta-predicate *call(+Term)* behaves as if *Term* appears textually in *call(+Term)*'s place. The predicate *rcall(+Channel,?Term)*, is a meta-predicate that behaves exactly like *call* but executes against a remote Prolog database. The *rcall* predicate takes two arguments. The first argument is the address of the computer to execute a predicate on. The second argument is the goal to be executed. The idea is simple. If the programmer knows how to use *call*, the programmer will know how to use *rcall*.

These remote predicates are based on the client server model and therefore the complementary server predicate for *rcall* is *listen_to(+Port)*. A prerequisite to using *rcall* is that a Prolog server must make its Prolog database available by executing *listen_to*. *listen_to* prepares the server for acceptance of remote predicates and blocks until a remote predicate is submitted. Upon receipt of a predicate, the server evaluates it and returns the result to the waiting client. When the session is over, the client terminates the conversation, the server resets itself and then waits for predicates from other Prolog clients.

As an example, assume that there is a Prolog database on a remote computer called *csr* with the following database of Prolog facts

```
cat('Muffin').
cat('Frodo').
cat('Rub-A-Dub').
```

Further, assume that *csr* has made its Prolog database available by executing the predicate

```
?- listen_to(2001).
```

To ask questions of *csr's* Prolog database the client executes the query

```
?- rcall(port(csr,2001),cat(X)).
```

and as expected, Prolog responds with

```
X=Muffin
```

followed by

```
X=Frodo;
X=Rub-A-Dub
```

should subsequent answers be requested

There can be several occurrences of *rcall* in a query, all referring to the same or different servers For example, it is perfectly acceptable to perform the following query

```
?- rcall(port(csr,2001),cat(X)),
   rcall(port(csr,2001),cat(X)).
```

and as expected, Prolog would respond with

```
X=Muffin
```

followed by

```
X=Frodo;
X=Rub-A-Dub[4]
```

---

[4]*rcall*'s ability to backtrack has also been tested by remotely executing other predicates such as *conc/3* as defined in [BR90], page 72

```
listen_to(Port_num) :-
      see(port(self,Port_num)),
      read(Term),
      call(Term),
      tell(port(self,Port_num)),
      write_canonical(Term).

rcall(Port,Term) :-
      tell(Port),
      write_canonical(Term),
      see(Port),
      read(Answer),
      Term = Answer.
```

Figure 14 Basic Implementation of Remote Predicates

4.2 The Basic Implementation of Remote Predicates

Figure 14 shows the basic implementation of remote predicates *Listen_to* waits passively for a client to send it a term When the client connects it wakes up the server and sends it a term to evaluate The server evaluates the term and returns the result to the waiting client

This implementation is basic because it lacks three important capabilities of the *call* predicate, namely the ability to obtain more than one answer to a query, the ability of the Prolog client to display server-based *user_output* and the ability of the Prolog server to obtain client-based *user_input*

Solutions to these problems are the topic of the following sections

4.3 Dealing With Failure

The problems associated with implementing distributed backtracking can be divided into two distinct parts communicating failure to the client when the server fails to evaluate a predicate sent to it, and enabling the *rcall* predicate itself to backtrack when this failure occurs

4.3.1 Communicating Failure

The problem of communicating failure is solved by introducing a communication protocol for communication between the client, *rcall*, and the server, *listen_to*. The state diagrams for this protocol is shown in figures 15 and 16. This protocol consists of messages called protocol atoms that prefix terms sent between the client and server. Valid protocol atoms are: *success*, *failure*, *termComing*, *anotherAnswer*, *charComing*, *charRequired* and *exitGoRemote*.

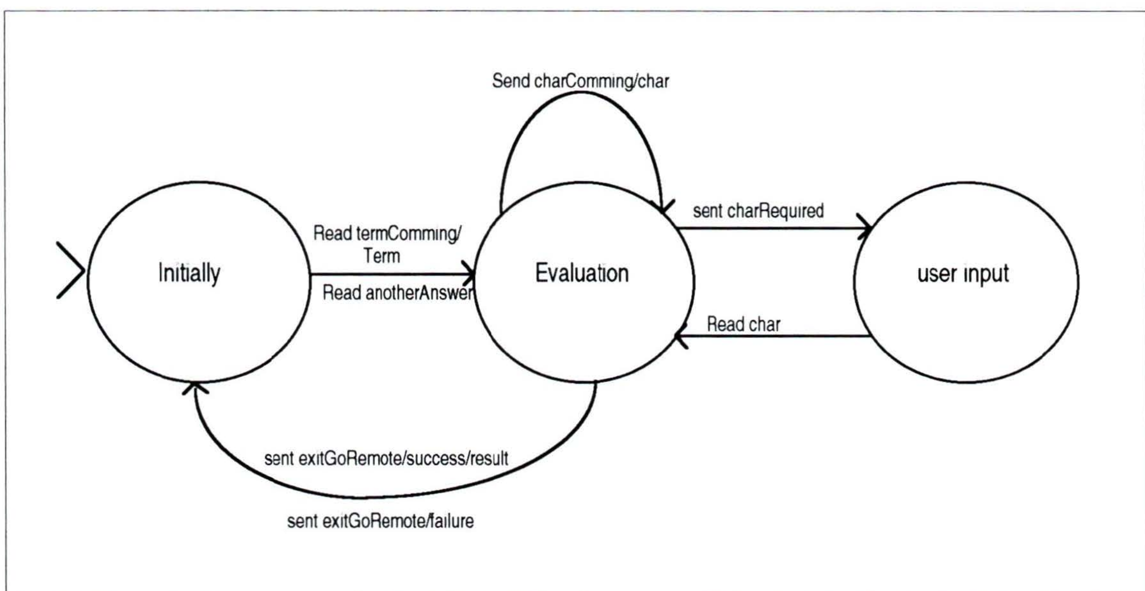


Figure 15. Server State Transition Diagram

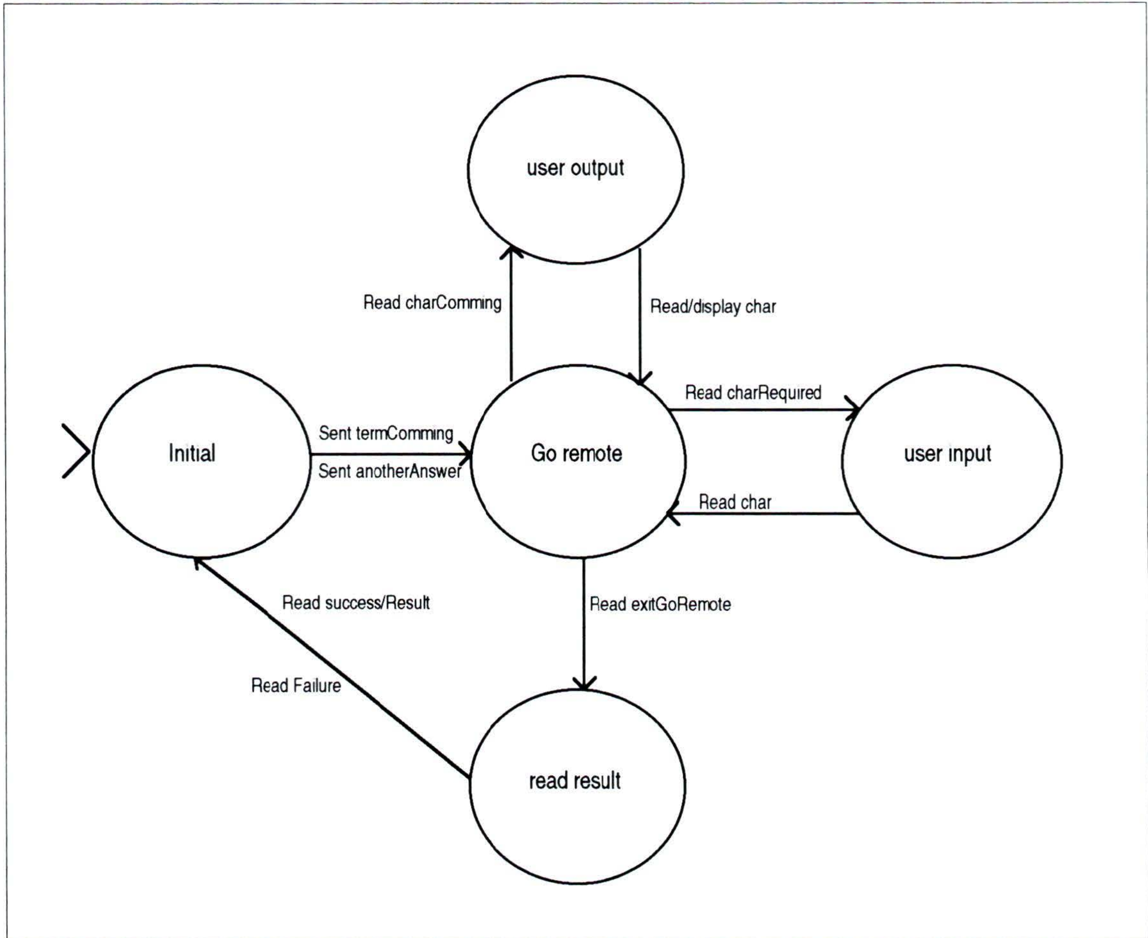


Figure 16. Client State Transition Diagram

To use this protocol, the client and server must prefix all terms communicated with the protocol atoms. For example, before sending a term for evaluation, the client sends the server the protocol atom *termComing*, indicating it is about to send a new query. When the client wants to backtrack, it sends the server the term *anotherAnswer*, indicating it would also like the server to backtrack and search for another answer.

If the server is successful in evaluating the predicate, it sends the protocol atom *success* followed by the result. In the event of failure, the server returns the protocol atom *failure* indicating it was unable to evaluate the predicate.

The protocol atoms *charRequired* and *charComing* are used for redirecting *user_input* and *user_output* and are discussed in detail in section 4 4

4.3.2 Implementing Distributed Backtracking

Implementing distributed backtracking presents some problems To see why, consider the following facts residing on the server

```
teacher(john).
teacher(jack).
teacher(mary).
```

If the query

```
?- rcall(port(csr,2001),teacher(X)).
```

were to be executed by a client using the basic implementation of *rcall* given in figure 14, the client would correctly receive the answer

```
X=john
```

However, if the client's own user subsequently asked for an additional answer, the system would respond incorrectly with

```
no.
```

The reason for this behavior is a result of the two distinct Prolog systems evaluating two different Prolog predicates The server evaluates the teacher predicate and the client evaluates the *rcall* predicate As such, the choice point for the teacher predicate resides on the server No corresponding choice point exists on the client, therefore when an additional answer is requested by the Prolog user, the client is unable to provide it

This problem can be solved by ensuring that the *rcall* predicate creates a client-based choice point that corresponds to the server-based choice point However, it is impossible for the client to mimic the choice point of the server exactly since the server's database will not in general be the same as the client's

```
repeat.
repeat :- repeat.
```

Figure 17 Prolog Definition of Repeat

Instead, the client's choice point must persist until the server indicates that there are no more answers to be had

The behavior of the client-based choice point is slightly different from a standard choice point While choice points typically give Prolog the ability to explore alternate paths of execution by executing different clauses with the same name, this choice point would evaluate the same clause upon backtracking The consequence is that *rcall* must be implemented using a *repeat*-like construct where the definition of repeat is shown in figure 17

Using *repeat*, the basic implementation of *rcall* may be enhanced to include distributed backtracking as shown in figure 18

Using this implementation, *rcall* will continue to backtrack until the server sends the client the protocol atom *failure* To prevent further backtracking following reception of *failure*, the Prolog cut (!) is used to remove the choice point created by *rcall* or *repeat*

4.4 Redirection of user_input and user_output

Two operational issues regarding the handling of *user_input* and *user_output* were addressed in the design of remote predicates To allow existing Prolog programs that contain embedded user-I/O to be used on the server unchanged, server-based reads from *user_input* and server-based writes to *user_output* should be redirected to the client This behavior is partially achieved using the I/O redirection capabilities introduced with chapter three's Channel

```
write_remote(Term) :-
      write_canonical(Term),
      write('.'),
      nl.

rcall(Port,Term) :-
      see(Port),
      tell(port),
      write_remote(termComing),
      write_remote(Term).
      read(Status),
      (Status == failure,!,fail
       ;
       read(Answer); Term = Answer
      ).

rcall(Port,Term) :-
      repeat,
      write_remote(anotherAnswer),
      read(Status),
      (Status == failure,!,fail
       ;
       read(Answer); Term = Answer
      ).
```

Figure 18 Basic *rcall* Enhanced With Repeat

I/O model The remainder is provided using a client-based predicate called *go_remote*

Redirecting I/O to and from the communication channel between the client and server is of little use by itself as there must be a client function to process the redirected user-I/O Server based *user_output* needs to be displayed on the client computer and server-based *user_input* needs to be obtained from the client computer on behalf of the server This is the function of the *go_remote* predicate

The *go_remote* predicate processes four protocol atoms, *charComing*, *charRequired*, *ungetChar* and *exitGoRemote* When the server writes to *user_output*, it prefixes each character to write with the protocol atom *charComing* and sends

```
read_remote_char(Char,Client) :-
      seeing(CurInp),
      see(Client),
      read_ch(Char),
      see(CurInp).

write_remote_char(Char,Client) :-
      telling(CurOutp),
      tell(Client),
      write_ch(Char),
      tell(CurOutp).

unget_remote_char(Char,Client) :-
      seeing(CurInp),
      see(Client),
      unget(Char),
      see(CurInp).

go_remote(Server) :-
      repeat,
      read_remote_char(Cmd,Server),
      process_remote_cmd(Cmd,Server).


process_remote_cmd('x',_).

process_remote_cmd('r',Server) :-
      read_remote_char(Ch,user_input),
      write_remote_char(Ch,Server),
      fail.

process_remote_cmd('w',Server) :-
      read_remote_char(Ch,Server),
      write_remote_char(Ch,user_output),
      fail.

process_remote_cmd('u',Server) :-
      read_remote_char(Ch,Server),
      unget_remote_char(Ch,user_input),
      fail.
```

Figure 19. go_remote Implementation

the pair to the client for processing. Similarly, when the server requires a character from *user_input*, it requests one from the client by sending the client the

protocol atom *charRequired*. The client reads the character from the user and returns it to the server. When the server is ready to send the results of an evaluation to the client, it instructs the client to get ready for the result by sending it the *exitGoRemote* protocol atom. The placement of these protocol atoms with respect to the rest of the *rcall* protocol is shown in figures 15 and 16. The implementation of the *go_remote* procedure is shown in figure 19.

While it may seem inefficient to process user-I/O one character at a time, in reality, the inefficiency is not as severe as one might imagine. Firstly, only redirected user-I/O is processed a character at a time and since user-I/O is interactive, any additional processing time is hardly noticeable. Secondly, one might surmise that transmitting user-I/O one string at a time would be more efficient because it would decrease the number of packets transmitted. However, most implementations of TCP/IP buffer output until the output buffer is full or flushed. This buffering, will make the character-at-a-time strategy more efficient because individual characters are not actually sent one at a time which would result in many small packets with very little data content, but are buffered by the output buffering mechanism and therefore the packet's data content will be much higher, resulting in more efficient communication. Thirdly, processing user-I/O a character at a time allowed us to place user-I/O redirection within the channel subsystem (with the exception of the *go_remote* routine). The advantage of this placement is that new additions to the I/O library are implemented without concern for user-I/O redirection and will inherit the user-I/O redirection capability by default as a happy consequence of using the channel I/O routines.

4.5 Prolog Implementation of the Remote Predicates

Figure 20 shows the complete Prolog based implementation of remote predicates. Following the groundwork completed in the previous sections, the implementation is reasonably straight-forward, however, a few features of the implementation require comment.

The first is that the server must be careful to preserve the current input and output streams between reads from and writes to the communication channel set up between the client and server. If the server did not do this, redirection of server-based user-I/O would not function properly. For example, if the current input stream on the server is *user_input* and the server changed the current input stream to be from the channel between the client and server and subsequently did not change it back to when it was finished with the channel, the next time a server-based read occurs, it will be from that communication channel, not from *user_input* as was originally intended. This situation is avoided using the *read_remote_term* and *write_remote_term* helper predicates.

Secondly, many TCP/IP implementations buffer output to improve efficiency because the greater the data content of a packet, the more efficient the transmission. However, there are occasions when a term needs to be sent immediately, otherwise, deadlock may occur. For example, consider a server that requires input from the user. It sends the *charRequired* protocol atom to the client and blocks, waiting for the response. The client on the other hand is waiting in the *go_remote* predicate for instructions from the server. These instructions never arrive because the protocol atom *charRequired* is still in the partially full output buffer on the server and both the client and server wait indefinitely for input from each other. To avoid this problem, the built-in

predicate *flush* is used to immediately send any buffered output, for which an immediate response is required

Thirdly, broken network connections or failed clients and servers are detected by the *end_of_file* term returned by the *read* predicate When a client breaks a connection, the lower level implementation of TCP/IP returns *end-of-file* to the upper level channel sub-system This action seems to be adequate for this implementation of remote predicates but the unhappy side effect of this is that the remote predicates are unable to distinguish between a failed evaluation and a broken network connection A better solution would be for the remote predicate to raise an exception using an exception handing scheme such as *catch* and *throw* upon detecting this end of file condition as proposed by the ISO standard for Prolog The current implementation of TOPIC does not have this mechanism

The query *catch(Goal1,Arg,Goal2)* is like *call* except that if at any stage during the execution of *Goal1*, there is a *call* to *throw(Arg)*, then execution will immediately jump back to the catch and proceed to *Goal2* *Arg* need not be fully instantiated and therefore provides a convenient technique to communicate to *catch* what happened

This behavior may be exploited by our extended Edinburgh I/O model to handle I/O exceptions For example, calls to *read* may be written like

```
catch(read,read_error(TheError),handle_exception(TheError)).
```

If a read-error occurs, *read* will raise the exception *read_error* with the error that occurred communicated by the variable *TheError* The exception handler *handle_exception* will then be invoked to deal with this error condition

Finally, the client occasionally requires the servers it is currently communicating with to reset themselves, usually whenever the client resets

itself In the stand-alone interactive environment, this takes place at the end of a query, when no more answers are available or wanted To enable the clients to reset servers, the clients keep track of the channel addresses of servers they are communicating with Prior to the client resetting itself, the client sends the server the protocol atom *reset* The server in turn executes the *reset* built-in which resets the server's inference engine to an initial state but leaves all existing communication channels intact

```prolog
% HELPER predicates

% once(?Term)
%
% Execute a predicate exactly once
once(Term):-
      call(Term),
      !.

% set_current_io(+Inp,+Outp)
%
% set current input and output
set_current_io(Inp,Outp):-
      see(Inp),
      tell(Outp).

% get_current_io(-Inp,-Outp)
%
% Get current input and output stream names
get_current_io(Inp,Outp):-
            seeing(Inp),
            telling(Outp).

% read_remote_term(-Term,+Port)
%
% read a term from the client read returns end_of_file when client breaks a
% connection
read_remote_term(Term,Port):-
      seeing(CurInp),
      see(Port),
      read(Term),
      see(CurInp).

% read_remote_char(-Term,+Port)
%
% read a character from the client read returns end_of_file when client breaks a
% connection
read_remote_char(Char,Port):-
      seeing(CurInp),
      see(Port),
      read_ch(Char),
      see(CurInp).
```

```
% write_remote_term(?Term,+Port)
%
% Write a term on a port, preserving the current output stream. Use write
%  canonical to ensure atoms are quoted. This predicate is used to send
% a Prolog term to a remote Prolog reader. Each term is written in its canonical
% form, followed by a full stop and a new line as is required by the read
predicate.
write_remote_term(Term,Port,Flusher):-
      telling(CurOutp),
      tell(Port),
      write_canonical(Term),
      write('.'),
      nl,
      call(Flusher),
      tell(CurOutp).

% write_remote_char(?Char,+Port,Flusher)
%
% Write a character on a port, preserving the current output stream. This
predicate
% is used to send a single character to a remote reader so that it may be read by
% read_remote_char
write_remote_char(Char,Port,Flusher):-
      telling(CurOutp),
      tell(Port),
      write_ch(Char),
      call(Flusher),
      tell(CurOutp).

% unget_remote_char(+Char,+Port)
%
% Returns character Char to input stream Port
unget_remote_char(Char,Port) :-
      seeing(CurInp),
      see(Port),
      unget(Char),
      see(CurInp).

% dont_flush
%
% dont_flush is passed as an argument to write_remote_term and
% write_remote_char when flushing of the output stream is not required.
dont_flush :-
      true.
```

```
% redirect(?Port)
%
% redirect current input and output to Port
redirect(Port):-
      redirect_input(Port),
      redirect_output(Port).

% redirected
%
% Unredirect input and output
redirected:-
      redirected_input,
      redirected_output.

%
% SERVER implementation
%

% listen_to(+PortNum)
%
% turn prolog session into prolog server
listen_to(PortNum):-
      get_current_io(CurInp,CurOutp),
      Port=port(self,PortNum),
      assert(self(Port)),
      set_current_io(Port,Port),
      set_current_io(CurInp,CurOutp),
      redirect(Port),
      do_remote(Port).

% do_remote(+Port)
%
% read action and start processing. Valid actions are
% termComing,end_of_file,anotherAnswer and reset
do_remote(Port):-
      read_remote_term(Action,Port),
      deal_with_client(Action,Port).

% deal_with_client(+Action,+Port)
%
% Port intends to send a term
deal_with_client(termComing,Port):-
      read_remote_term(Term,Port),
      evaluate(Term,Port).
```

```
% Port wants another answer, backtrack to last evaluation and get another
% answer
deal_with_client(anotherAnswer,_):-
	fail.

% Clear all choice points and start again
deal_with_client(reset,port(self,PortNum)):-
	reset(start_over).

% Port broke connection
deal_with_client(end_of_file,_):-
	redirected,
	halt.

% evaluate(?Term,+Port)
%
% Client broke connection, wait for another connection. This worked under
%  UNIX because see(port(self,PortNum)) forked another process. Halt would
% kill the child process. The Mac and PC do not fork processes so we have to
% find something different to do. Attempt to evaluate the Term
evaluate(end_of_file,_):-
	halt.

% Successful evaluation
evaluate(Term,Port):-
	call(Term),
	cancel_go_remote(Port),
	write_remote_term(success,Port,dont_flush),
	write_remote_term(Term,Port,flush),
	do_remote(Port).

% Evaluation failed. Indicate failure to server and  wait for further actions.
% Last-Call optimization is  assumed therefore the call to do_remote will not
% eventually blow the stack.
evaluate(_,Port):-
	cancel_go_remote(Port),
	write_remote_term(failure,Port,flush),
	do_remote(Port).

% cancel_go_remote(+Port)
%
% Client currently waiting in go_remote. cancel_go_remote informs the client
% results are  forthcoming.
```

```
cancel_go_remote(Port):-
      write_remote_char('x',Port,dont_flush).

% start_over
%
% First predicate executed when server is reset
start_over:-
      self(Port),
      do_remote(Port).

%
% CLIENT implementation
%

% talk_to(+Server)
%
% Can I talk to the server? Open a connection to Server but
% preserve current I/O
talk_to(Server):-
      get_current_io(Inp,Outp),
      set_current_io(Server,Server),
      set_current_io(Inp,Outp).

% rcall(+Server,?Term)
%
% Set up remote connection, send remote term to  Server and wait for result
rcall
% may fail in two cases, when the communication fails, or when evaluation fails
rcall(Server,Term):-
      talk_to(Server),
      once(register_server(Server)),
      do_rcall(Server,Term).

% do_rcall(+Server,?Term)
%
% Send ?Term to +Server for evaluation
do_rcall(Server,Term):-
      write_remote_term(termComing,Server,dont_flush),
      write_remote_term(Term,Server,flush),
      once(go_remote(Server)),
      deal_with_server(Status,Term,Server),
      (Status==failure,!,fail;true).
```

```
do_rcall(Server,Term):-
      get_another_answer(Server,Term).

% get_another_answer(+Server,?Term)
%
% Get additional answers from server
get_another_answer(Server,Term):-
      repeat,
      write_remote_term(anotherAnswer,Server,flush),
      once(go_remote(Server,Cmd,Ch)),
      deal_with_server(Status,Term,Server),
      (Status==failure,!,fail;true).

% deal_with_server(+Status,?Term,+Server)
%
% deal_with_server and process_status processes RCall protocol. The following
% status' may be returned from the server: success,failure,end_of_file.
deal_with_server(Status,Term,Server):-
      read_remote_term(Status,Server),
      process_status(Status,Term,Server).

% process_status(+Status,?Term,+Server)
%
% success
process_status(success,Term,Server):-
      read_remote_term(Answer,Server),
      Term=Answer.

% failure
process_status(failure,_,_).

% Server broke connection
process_status(end_of_file,_,Server):-
      close_connection(Server).

% go_remote(+Server)
%
% User I/O redirection handler
go_remote(Server) :-
      repeat,
      read_remote_char(Cmd,Server),
      process_remote_cmd(Cmd).
```

```
% process_remote_cmd(+Cmd)
%
% handle the various protocol atoms to do with user-I/O redirection
process_remote_cmd('x').

process_remote_cmd('r') :-
      read_remote_char(Ch,user_input),
      write_remote_char(Ch,Server,flush),
      fail.

process_remote_cmd('w') :-
      read_remote_char(Ch,Server),
      write_remote_char(Ch,user_output,flush),
      fail.

process_remote_cmd('u') :-
      read_remote_char(Ch,Server),
      unget_remote_char(Ch,user_input),
      fail.


% register_server(+Server)
%
% Record the existence of an active rcall server. These facts are used by
% reset_servers.
register_server(Server):-
      is_server(Server).
register_server(Server):-
      assert(is_server(Server)).


% reset_servers
%
% sent reset command to all rcall servers
reset_servers:-
      is_server(Server),
      write_remote_term(reset,Server,flush),
      fail.

reset_servers.


% close_connection(+Server)
%
% Close network connection opened by rcall
close_connection(Server):-
      is_server(Server),
      telling(Who),
      seeing(What),
```

```
     tell(Server),
     told,
     see(Server),
     seen,
     retract(is_server(Server)),
     tell(Who),
     see(What).

close_connection(Server) :-
     write(Server),
     write(' is not an rcall server.'),
     nl.
```

Figure 20 The Implementation of RCall

4.6 Technical Issues

A number of technical issues were addressed during this project The four most important of these were issues dealing with marshaling of data between the client and server, handling multiple simultaneous clients on the server, the choice of programming languages (C or Prolog) to implement the remote predicates with, and dealing with client-based cuts within distributed Prolog programs These issues will be discussed in the following sections

4.6.1 Marshaling of Data

Early in this project, it was recognized that it would be convenient to use the built-in term reading and writing capabilities of the existing Prolog system to transmit terms between processes This capability would solve a problem that exists in the standard remote procedure call, namely marshaling of data between processes Marshaling refers to packaging and transmitting data between two processes Marshaling involves two activities flattening of structured data into messages for transmission and converting data such as floating point numbers into an agreed upon format

Flattening a data structure is complicated and sometimes requires the hand coding of the marshaling routine to handle special cases The complexity is evident when the marshaling routine must deal with pointers to data Clearly, because a pointer refers to data at an address in the computer's memory space, and because processes communicating via remote procedure calls do not share memory, the marshaling routine must decide how to handle pointers One strategy would be to de-reference the pointer and copy the data the pointer points to This strategy while having merit, is not without its problems For example, consider a pointer to a node in a linked list Should the marshaling routine copy the node, the rest of nodes in the list from that node, or the entire list? A case may be made for either of these solutions Many implementations of remote procedure calls side-step this problem by prohibiting passing pointers as arguments to remote procedures

The need to perform data conversion arises from the problem of heterogeneity Different computers represent similar data differently For example, an integer on one computer may be represented in Little-Endian format, where the least significant byte of an integer has the highest address Other computers represent integers in Big-Endian format where the least significant byte of an integer has the lower address A similar problem exists for real numbers Different computers may represent real numbers differently If two computers need to exchange such data, they must agree on a standard representation for that data

This problem is solved by transmitting terms between computers using their string representations Although this may be less efficient than transmitting an internal representation of the terms, it is an acceptable price to pay given the complexities involved Any inefficiencies realized will not be due to the slightly

larger amount of data transmitted but will be due to the extra parsing of terms, and the conversion to and from decimal characters by the Prolog client and server On high-speed communication networks such as ethernet, the time taken to transmit a packet is fairly constant across message sizes A packet that is only a few bytes long takes as long to transmit as one that is a few kilobytes long because most of the overhead in inter-process communication is in processing the network protocol

4.6.2 The Server's Ability to Handle Multiple Clients

Most servers in a typical client-server application have the ability to handle multiple clients simultaneously Under UNIX, this is often handled by process forking where the server creates a new process each time a client connects to a server The new process handles the client's request while the original process goes back to waiting for other clients

While handling multiple clients is easily implemented under multi-tasking operating systems such as UNIX, implementation is more challenging on the other platforms that do not readily support multiple processes within a single application such as IBM PC's or Macintosh computers

A number of solutions have been proposed for this problem For example, IC Prolog II [CC93] implements its own process structure Each process is represented by a separate process control block consisting of the WAM structures STACK, HEAP and TRAIL, and a set of WAM registers containing the register values at the time of the context switch While this solution has advantages over process forking in terms of efficiency of process creation, on platforms such as the IBM PC or the Macintosh where memory is a scarce resource, this strategy may not be suitable The difficulty is that this strategy

requires each WAM process to have a large static storage area. The actual space used within these structures varies from program to program and a-priori knowledge of each program's space requirement is unknown, and is unlikely to approach the maximum allocation. Hence, utilization tends not to be very high.

A more attractive solution would be to devise some scheme where processes could share the various WAM based data structures. While the details have not been worked out in the presence of garbage collection, it should be possible to share the largest WAM structure, the HEAP.

4.6.3 Choice of Programming Language

The first attempt at implementing remote predicates was performed mostly in C but this implementation was subsequently discarded in favor of a mostly Prolog implementation. The reason for this is simple. As the requirements were more fully realized and the necessary interactions between the WAM and the remote predicates grew, the implementation became very complicated and it was realized that, in the end, very little confidence could be had in the correctness of the C based implementation. A method to control this complexity was needed. A Prolog implementation proved to be the answer.

Although a mostly Prolog implementation was clearly a better choice, a few ancillary functions written in C were needed, namely the channel functions and the *reset* predicate.

4.6.4 Dealing with Cut Within Client Programs.

Cut (!) is used in Prolog programs to prevent backtracking. When a cut is encountered in a Prolog program, Prolog discards all alternative choices between the time the parent goal was invoked and the time the cut was

encountered The parent goal is the goal that matched the head of the clause containing the cut For example consider the following program

```
a(1).
a(2):-!.
a(3).
```

and the query

```
?- a(X).
```

This program would produce two answers, X=1 and X=2 Upon encountering the cut in the second clause, Prolog throws away all alternate paths of execution

An apparent problem arises with cuts occurring in client programs using remote predicates To see the problem, consider the following program residing on a Prolog server *csr* waiting on application port *2001*

```
cat(muffin).
cat(frodo).
cat(rub_a_dub).
dog(rover).
dog(spot).
snake(sam).
```

and the following query occurring on the client

```
a(X) :- rcall(port(csr,2001),cat(X)),write(X),nl,b.
b :- rcall(port(csr,2001),dog(Y)),!,write(Y),nl,
        rcall(port(csr,2001),snake(Z)),write(Z),nl.
?- a(X).
```

The query a(X) should result in the following output

```
muffin
rover
sam
X=muffin;

frodo
rover
sam
X=frodo;
```

```
rub-a-dub
rover
sam
X=rub-a-dub
```

In reality, only the first answer is given This behavior is a result of a cut occurring in the **b/0** predicate When the cut is encountered, the client-based choice point created by the **rcall(port(csr,2001),dog(Y))** goal is thrown away but the server-based choice point for **dog(Y)** remains When failure is induced by the user typing a ';', the client backtracks to the `rcall(port(csr,2001),cat(X))` in the `a/1` predicate and sends the server the protocol atom `anotherAnswer` The server then backtracks to the second `dog` clause, not knowing a client-based cut has occurred The server generates the second answer for `dog` and returns it to the client The client then attempts to unify the answer with the term `cat(X)` and fails, and therefore assumes no more answers are to be had The problem is how to keep the client and server synchronized when a cut occurs

This problem could be solved by adding special code to simulate on the server the operations that occur on the client during a cut The implementation of cut with respect to the WAM is described in detail in [HA91] Briefly, it involves discarding all the choice points that were created after the choice point frame that was current right before the call to the procedure containing the cut

This behavior is accomplished by preserving the value of the WAM's choice point register *B* at the time of the procedure invocation and resetting *B* to this value, when the cut occurs [5]

[5]Executing a cut also involves cleaning up another WAM structure called the *trail* that contains locations of variable bindings that are conditional on successful execution of a predicate

The Prolog compiler is required to examine each clause and determine whether or not it contains a *cut* predicate. If it does, it emits the instruction *get_level* that preserves the current value of *B*. When the cut instruction is executed, this saved value is re-assigned to *B*.

To mimic this behavior in the distributed environment, it may be possible to have the Prolog compiler also generate a *remote_get_level* instruction that would cause the affected remote servers to preserve their *B* registers in preparation for a cut to occur. When a cut occurs, the cut predicate could send the affected servers a protocol atom *cut* which would cause each of the servers to execute a *remote_cut* predicate that would remotely reassign the *server's* B register.

Chapter 5

An Example Application Using Remote Predicates

Remote predicates can be used for a wide variety of applications and are particularly well suited to client-server type applications. To illustrate their effectiveness, a document delivery system in the spirit of Gopher was developed.

Gopher was developed in 1991 at the University of Minnesota primarily to act as a distributed document delivery system. The Gopher protocol is very simple and is designed to facilitate efficient implementation on low-end TCP/IP capable workstations such as IBM PCs or Macintosh computers. The Gopher protocol is stateless, that is, when a Gopher-client connects to a Gopher-server, the server sends the client a block of text containing the information that the server can provide, complete with the addresses of where to obtain this information. Following this exchange, the connection is broken. No intermediate information is retained by the server. This behavior limits conversations to short bursts of information, thus allowing a Gopher server to service a large number of clients and frees the server from having to manage multiple network connections.

The Prolog code for our Gopher-like application is shown in figures 21 and 22. The application consists of a Gopher client and many Gopher servers. The client contacts the server and obtains a list of items about which the server can provide information. The client displays this list to the user and the user then makes a selection. Each item in this list is one of three types: a document, a sub-menu or another server. If the user selects a document, the document is displayed to the user. If the user selects a sub-menu, the client queries the server

for the list of items on the sub-menu and then displays this list to the user. If the item is another server, the client then contacts the server and obtains that server's item list.

The menu items are stored on each server in a file. The file consists of Prolog terms of the form *menuItem(type,fileName,label)*. Type may be one of *gopher,document,subMenu*. The file name is either the network address of another server, the file name of a document or the name of a file containing the sub-menu.

This application shows how well suited Prolog with the addition of remote predicates is for writing client-server applications. The server program is about one half page of Prolog and most of the code implements the menu file. The client is about one and a half pages of Prolog and mostly deals with displaying menu items and obtaining user selections. This program would be even shorter if exception handling were a part of this Prolog system.

The original gopher program was written in C and is quite large. Notwithstanding that much of that program is concerned with the user interface, it is hard to imagine that it could be written in 2 pages of C.

```
gopher(GopherServer) :-
      rcall(GopherServer,getRootMenu(TheMenu)),
      close_connection(GopherServer),
      doGopherStuff(GopherServer,TheMenu).

doGopherStuff(TheServer,TheMenu) :-
      displayMenu(TheMenu),
      getSelection(TheMenu,TheSelection),
      processSelection(TheServer,TheMenu,TheSelection).

displayMenu([]).

displayMenu([menuItem(Type,Address,Label) | Rest]):-
```

```
    write('['),
    write(Type),
    write(']'),
    write(': '),
    write(Label),
    nl,
    displayMenu(Rest).

getSelection(TheMenu,TheSelection):-
    write('Select: '),
    read(TheLabel),
    findLabel(TheSelection,TheMenu,TheLabel).

getSelection(TheMenu,TheSelection):-
    write('Invalid selection.'),
    nl,
    getSelection(TheMenu,TheSelection).

findLabel(_,[],_) :-
    fail.

findLabel(TheSelection,

[menuItem(TheType,TheSelector,TheLabel)|_],TheLabel):-
    TheSelection = menuItem(TheType,TheSelector,TheLabel).

findLabel(TheSelection,[_ | Rest],TheLabel) :-
    findLabel(TheSelection,Rest,TheLabel).

processSelection(TheGopher,TheMenu,
        menuItem(gopher,GopherAddr,_)) :-
    gopher(GopherAddr).

processSelection(TheGopher,TheMenu,
        menuItem(subMenu,SubMenuName,_)) :-
    rcall(TheGopher,getMenu(SubMenuName,TheSubMenu)),
    close_connection(TheGopher),
    doGopherStuff(TheGopher,TheSubMenu).

processSelection(TheGopher,TheMenu,
        menuItem(document,DocumentName,_)) :-
    rcall(TheGopher,see(DocumentName)),
    rcall(TheGopher,gets(TheStr)),
    copyRemoteFile(TheGopher,TheStr),
    rcall(TheGopher,seen),
    close_connection(TheGopher),
    doGopherStuff(TheGopher,TheMenu).

processSelection(TheGopher,TheMenu,menuItem(halt,_,_)) :-
```

```
     halt.

copyRemoteFile(TheGopher,end_of_file).

copyRemoteFile(TheGopher,TheStr) :-
     write(TheStr),
     nl,
     rcall(TheGopher,gets(AnotherStr)),
     copyRemoteFile(TheGopher,AnotherStr).
```

Figure 21 Prolog Based Gopher Client

```
%
gopherServer :-
     listen_to(2001).

getRootMenu(TheMenu) :-
     getMenu('gopher.mnu',TheMenu).

getMenu(MenuName,TheMenu) :-
     see(MenuName),
     buildMenu(TheMenu),
     seen.

buildMenu(ItemList) :-
     read(Item),
     buildItemList(Item,ItemList).

buildItemList(end_of_file,[]).

buildItemList(Item,[Item | Rest]) :-
     read(AnotherItem),
     buildItemList(AnotherItem,Rest).
```

Figure 22 Prolog Based Gopher Server

Chapter 6

Related Work

This chapter describes two projects that are related to the remote predicates implemented in this thesis. The first is IC-Prolog II [CC93] implemented by Damian Chu and Keith Clark. The second is Delta-Prolog implemented by Luis Moniz Pereira and Roger Nasr [PN84].

6.1. IC-Prolog II

IC-Prolog II (ICP for short) is an implementation of Prolog containing a number of extensions that facilitate writing distributed applications. Support for concurrent execution and communication is central to ICP. ICP contains primitives that support both fine and coarse grained concurrency as well as three mechanisms for performing inter-process communication: Pipes, TCP/IP sockets and mail-boxes.

ICP is actually two programming languages: Prolog and Parlog. Prolog is able to call Parlog and Parlog is able to call Prolog through the use of the meta-predicates parlog/1 and prolog/1 respectively. The reason for including a Parlog sub-system is that while ICP supports coarse-grained parallelism, Parlog supports fine-grained parallelism - the kind of parallelism that is inherent in many logic-based programs. Such programs typically create a large number of short lived threads and the overhead of creating a large number of coarse grained processes becomes significant.

Parallelism in Parlog is a result of the parallelism that is inherent in many logic programs. Parlog programmers do not explicitly state which parts of a program are run in parallel. This is taken care of by the Parlog language itself. Communication between Parlog processes is also implicit.

Parlog supports stream-AND parallelism and Committed-OR parallelism. Stream AND parallelism is the concurrent evaluation of calls which share variables, with the value of the variable communicated incrementally and implicitly between the calls. A goal requiring a result from a previous goal blocks until that variable is instantiated. This forms a kind of producer-consumer relationship between the two goals. A feature of most languages using stream-AND parallelism is that the programmer is required to state explicitly which arguments of each predicate are input and which are output.

Committed-OR parallelism is the parallel search for candidate clauses. When a match is found, the search is terminated and the evaluation commits to that clause. This is in contrast to Prolog where, because of committed choice, Parlog programs do not backtrack. Additionally, while Prolog programs are able to provide all possible answers to a query, Parlog programs are not.

Coarse-grained parallelism is provided within ICP to facilitate the handling of multiple requests from other processes simultaneously. Such a capability is often needed in client-server applications. ICP's support for multiple threads is provide by a built-in primitive called fork/1 with the single argument to fork being the predicate to execute in the new process. Each ICP process is a distinct WAM-like structure consisting of a STACK area and a set of WAM registers. To prevent a single process from monopolizing the processor, ICP implements pre-preemptive multitasking between threads.

Communication between threads executing on the same processor occurs using a uni-directional structure called a pipe. A pipe contains two communication ports, one for writing to and the other for reading from. Typically, a Pipe is created using the pipe/2 built-in predicate, and then two processes are forked that will make use of the pipe.

```
?- pipe(Out,In),
     fork(server(In)),
     fork(client(Out)).

server(In) :-
     read_pipe(In,Req),
     fork(service(Req)),
     server(In).
```

Figure 23 Client-Server Skeleton in IC-Prolog II

An example of how the server portion of a client-server application may be implemented in IC-Prolog II is shown in figure 23 The use of fork for servicing a request prevents a client's request from monopolizing the processor as well as allowing the server to service additional requests from clients immediately without having to wait for the current request to complete

Communication between processes residing on different platforms may be accomplished in ICP using one of two other methods - either TCP/IP primitives or a higher level mechanism implemented on top of TCP/IP called the mail-box The TCP/IP primitives closely resemble the UNIX interface to Berkeley sockets The main advantage of using TCP/IP is that these primitives may be used to interact with other TCP/IP applications such as a UNIX mail daemon or Gopher servers

Obvious similarities exist between the extensions to Prolog implemented in thesis and IC-Prolog II Both implement communications primitives based on TCP IC-Prolog II made its interface to TCP rather specific, our extensions were implemented as a simple extension to the Edinburgh I/O model Both implementations support the coarse-grained concurrency required for client-server applications ICP's multi-threading capability is explicit through the use

of the fork primitive Our's is implicit with the fork occurring at the time a connection from a client occurs

Our support for client-server type applications is cleaner because communication between the client and the server is well hidden within the notion of the Prolog predicate Our implementation also allows for distributed backtracking between the client and server The tradeoff for backtracking is that the remote predicates are synchronous, although asynchronous execution is also possible using the extended Edinburgh I/O model where concurrently executing Prolog programs communicate using reads and writes to a network channel rather than through the use of remote predicates

6.2 Delta-Prolog

Delta-Prolog is a super-set of Prolog that approximates Distributed-Logic (Monteiro 1981-84) Without going into detail, distributed logic extends Horn-Clause Logic in two ways (1) by distinguishing between sequential and parallel composition of goals and (2) by introducing the time related notion of an event which provides for both process communication and synchronization

Events are represented using the notion of event-goals denoted by G ! E and G' ? E G and G' are arbitrary terms, E is a Prolog atom (called the event name), and ! and ? are binary predicate symbols G ! E may be reduced iff a complementary goal G'?E may be selected such that G and G' are unifiable G ! E and G' ? E execute in separate processes, possibly on different computers G ! E blocks until the complementary G'?E within a different process is executed Communication between the two processes occur when G and G' are unified The "?" and "!" operators are similar to Hoare's input and output commands found in his landmark paper on Communicating Sequential Processes [HO78]

(process 1)

```
squares :- write(0), nl, sq(0)

sq(Q) :- I ? mail,
         R is Q + I, write(R), nl,
         sq(R).
```

(process 2)

```
odds :- odd(1).

odd(I) :- I ! mail,
          J is I+2, write(I), nl,
          odd(J).
```

Figure 24 Squares Example in Delta-Prolog

This I/O mechanism provides a convenient and powerful way of synchronizing processes and communicating between processes

As a simple example of a Delta-Prolog program, consider one that computes squares according to the formula $K^2 = (K-1)^2 + (2K-1)$ for $K>0$ Two processes, *squares* and *odds*, are launched simultaneously on two different terminals When *squares* begins, it waits at the goal **`I ? mail`** in the *`sq`* predicate until odds executes the **`I ! mail`** goal in the *`odd`* predicate When this happens, the *`I`* variable in the *`sq`* predicate is unified with 1 and within the squares process continues with a recursive call to **`sq/1`**, where it again waits for the odds to execute the **`I ! mail`** goal

The relationship of Delta-Prolog to our remote predicates may not be immediately obvious but using its distributed logic primitives, it is possible to implement remote predicate as is shown in the following figure

(Server)

```
Job :- repeat, go.

go :- launch(G) ? Job,
     (G, solution(G) ! Job,
      Option ? Job,
          ( Option == reset, !, fail ;
            Option == halt, halt;
            Option == backtrack, fail );

      solution(fail) !  Job, fail ).
```

(Client)

```
launch(G,Job) :-
     launch(G) ! Job;

     solution(S) ? Job,
     (S == fail; reset ! job), !, fail.

solution(G, Job) :-
     repeat,
     solution(S) ? Job,
     (S == fail, !, launch(G) ! Job, fail;
      S = G;
      backtrack ! Job, fail ).
```

Figure 25. Remote Predicates in Delta-Prolog

After some study, one might draw the conclusion that *launch* is essentially the same as the *rcall/2* predicate and *Job* is essentially the same as the *listen_to*/1 predicate, however, there are some important differences.

Firstly, Delta-Prolog's implementation does not support multiple remote calls to the same server within the same query. For example, using Delta-Prolog, it would not be possible to write a query of the form

```
?- launch(G,Job),
      solutions(G,Job),
      write(G),
      launch(H,Job),
      solutions(H,Job).
```

In contrast, using our remote predicates, This same code segment could be written

```
?- rcall(G,port(..)),
      write(G),
      rcall(H,port(..)),
      write(H).
```

Secondly, Delta-Prolog's remote predicates are not well suited to writing client-server applications because of their inability to handle queries from multiple clients simultaneously. Servicing a waiting client will not occur until servicing of the current client is complete. IC-Prolog II solves this problem using multiple thread primitives, but this solution would not work for Delta-Prolog. In Delta-Prolog communication takes place by unifying using "event goals". Recall that an event goal consists of an event name and a Prolog term. When a process executes an event goal, it blocks until a different process executes the complementary event goal with the same name. If a server process were to be created each time a client sends the server requests, it may be possible that events occurring in the concurrently executing servers would conflict. For example, consider two concurrently executing server processes forked from the same parent, waiting on event G!E to be executed by a client. When one client executes G?E, there would be no way of determining which server the event is intended for.

Thirdly, because our remote predicates are built on top of the Edinburgh I/O model using TCP/IP extensions, it is possible to interface our Prolog system to existing TCP/IP applications written in other programming languages.

Fourthly, location specific applications such as Gopher are easier to implement using our implementation of remote predicates. A gopher-client connecting to a Gopher server translates directly into a remote predicate call to a

particular computer on a particular port number Simulating this in Delta Prolog would be difficult

Finally, our remote predicates are somewhat unique in their handling of user I/O redirection Neither Delta-Prolog or IC-Prolog II have this feature

Chapter 7

Conclusions

In this thesis, we have investigated the merits of extending Prolog with primitives so that Prolog may be used for the development of client-server applications. These extensions are simple, both in implementation and in use. When considering the minimum amount of effort required to implement the example application, Gopher, it is easy to see that the availability of these extensions are of significant value.

The implementation was performed in two parts. The first was to extend an existing I/O model to include facilities for performing inter-process communication. The second was to write two library functions, *rcall* and *listen_to*, in Prolog that would implement the facilities necessary to allow Prolog predicates to be executed remotely. These predicates allowed not only for remote execution but remote user I/O redirection and distributed backtracking.

While other forms of remote execution are available, few are so simple. For example, the remote procedure call often requires hand-coding of marshaling routines in order to pass data back and forth. This complexity often prevents programmers from writing distributed applications. On the other hand, our remote predicates do not suffer from this limitation.

Our remote predicates also provide a convenient method of partitioning distributed applications around predicate boundaries. Many other distributed applications are centered around communication which often leads to programs that are difficult to understand. Communication in remote predicates is carefully hidden away from the programmer.

Our remote predicates were also designed to operate in a heterogeneous environment. This feature is necessary in today's computing environments as it

is not uncommon to see work groups consisting of many different kinds of computers. Our predicates have been successfully implemented on IBM PCs, Macintosh computers and UNIX workstations. Any one of these computers may function as a Prolog Client or a Prolog server.

APPENDIX A

Header file for the Channel Data Type

The following figure contains the fully commented header file for the Channel abstract data type introduced in chapter 3.

/* There are three different types of Channels. a_stream refers to standard file I/O. a_port refers to network I/O and a_either refers either standard or network I/O. */
```
typedef enum {a_stream,a_port,a_either} channelType;
```

/* channelDir indicates the direction of the channel. A channel is either an input channel or an output channel. */
```
typedef enum {a_input, a_output } channelDir;
```

/* The Channel structure contains pointers to 5 basic functions that a low level I/O system must provide. These functions pointers are assigned by an initialization function provided by the low level I/O system (i.e. StandardMakeChannel). handle is a file handle used to identify the open file during subsequent calls to the low level I/O subsystem. It also is assigned by the low level initialization function. */
```
typedef struct _channel{
```

/* read and return a character from input stream *handle*. Returns -1 on end-of-file. */
```
int (*read)(int handle);
```

/* Write character *c* on output stream *handle*. */
```
void (*write)(int handle, int c);
```

/* unread returns previously read character *c* to the input stream *handle* such that *c* will be returned if a subsequent call to read is made. */
```
void (*unread)(int handle,int c);
```

```
/* flush writes any buffered output to the stream identified by handle. The
channel remains open following a call to flush. */
void (*flush)(int handle);

/* close closes the file or network stream associated with handle. If the
stream has buffered output, the output is first flushed. */
void (*close)(int handle);

/* handle is the file handle returned from the call to the low-level I/O
initialization function. */
int handle;

/* The current cursor position use for user-streams. */
int line,col;

/* The last x coordinate of the cursor. */
int prev_col;

} channel;
```

```
/* AddChannel creates a new Channel with name n of type t in direction d using
channel structure c. The channel identifier is returned upon success. -1 is
returned if the name already appears in the channel table. -2 is returned if the
table is already full. AddChannel must be called before any other operations on
the channel are performed. */
extern int AddChannel(String n, channelType t,
                      channelDir d, channel c);

/* ChannelId returns the channel identifier of the channel called n of type t in
direction d. -1 is returned if the channel was not found. */
extern int ChannelId(String n, channelType t,
                     channelDir d);

/* DeleteChannel closes the channel identified by id and de-allocates any storage
allocated to it. */
extern void DeleteChannel(int id);

/* SetCurrentChannel sets the current channel to be channel id. A subsequent
call to CurrentChannel would return id. */
extern void SetCurrentChannel(channelDir d,int id);
```

/* CurrentChannel returns the channel id of the current channel in the direction of *d* */

```
extern int CurrentChannel(channelDir d);
```

/* Redirect redirects channel *source_id* such that input and output on it will actually be redirected to *target_id* */

```
Boolean Redirect(int source_id,int target_id);
```

/* ChannelName returns the name of the channel identified by *id* */

```
extern String ChannelName(int id);
```

/* ChannelType returns the channel type of the channel identified by *id* */

```
extern ChannelType ChannelType(int id);
```

/* ChannelFd returns the lower level I/O subsystem handle for the channel identified by *id* */

```
extern int ChannelFd(int id);
```

/* SetLink links two channel's cursors so that the correct prompt will be displayed when reading from user_input */

```
extern void SetLink(int id1,int id2);
```

/* Link returns the channel identifier of a channel linked to id It returns -1 if no channel is linked to *id* */

```
extern void Link(int);
```

/* SetPrompt defines a prompt to be used for user input */

```
extern void SetPrompt(char *);
```

/* GetPrompt returns the prompt set by SetPrompt NULL is returned if no prompt is defined */

```
extern char *GetPrompt(void);
```

/* xprintf writes formatted output on the current output channel The format string *f* follows the same formatting conventions as the C library function printf */

```
extern void xprintf(const int, char *f,...);
```

/* xputc writes the character c on the current output channel */

```
extern void xputc(char c);
```

/* xgetc reads a character from the current input channel -1 is returned in the end of file condition */

```
extern int xgetc(void);
```

/* xungetc returns character c to the current input channel A subsequent call to xgetc would return *c* */

```
extern void xungetc(const char c);
```

/* xgets reads a string into s from the current input channel xgets returns NULL on end of file condition */

```
extern char *xgets(char *s);
```

/* xputs writes string s on the current output channel */

```
extern void xputs(char *s);
```

/* xflush writes any buffered output associated with the current output stream on the current output stream */

```
extern void xflush(void);
```

/* xline returns the current line number of the cursor for the channel in direction d */

```
extern int xline(channelDir d);
```

/* xcol returns the current column number of cursor for the channel in direction *d* */

```
extern int xcol(channelDir d);
```

Commented Header File for Channel Subsystem

BIBLIOGRAPHY

[BA78] Backus, J "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs ", Communications of the ACM, vol 21, no 8, August, 1978, pp 613-641

[BR90] Bratko, I , "Prolog Programming for Artificial Intelligence, second edition", Addison-Wesley Publishing Company, 1990, pp 1-143, 533-560

[CC93] Chu, D and Clark, K , "I C Prolog II a Multi-threaded Prolog System", Technical Report, Imperial College of Science, Technology and Medicine, May, 1993

[CO93] Covington, M , "ISO Prolog A Summary of the Draft Proposed Standard", Technical Report, University of Georgia, 1993

[CR88] Carlton, M and Van Roy, P , "A Distributed Prolog System with And Parallelism", IEEE Software, January 1988, pp 43-51

[CS93] Comer, D and Stevens, D , "Internetworking with TCP/IP", Prentice Hall, 1993

[EN78] Enslow, Jr , P "What is a Distributed Data Processing System?", Computer, vol 11, January, 1978, pp 13-21

[GR87] Gregory, S , "Parallel Logic Programming in PARLOG", Addison-Wesley Publishing Company, 1987, pp 1-95

[HA91] Ait-Kaci, H , "Warren's Abstract Machine, A Tutorial Reconstruction", Mit Press, 1991

[HO78] Hoare, C A R , "Communicating Sequential Processes", Communications of the ACM, vol 21, no 8, August, 1978

[HL93] Horspool, N and Levy, H , "Translator-Based Multiparadigm Programming", Technical Report, University of Victoria, 1992

[HO92] Horspool, N , "The Berkeley UNIX Environment, second edition", Prentice Hall, 1992, pp 249-273

[KW92] ed Kacsuk, P and Wise, M , "Implementations of Distributed Prolog", John Wiley & Sons, 1992

[LH93] Levy, M and Horspool, N , "Translating Prolog to C a WAM-based approach", Technical Report, University of Victoria, 1993

[MU93] Mullender, S , "Distributed Systems, Second Edition", Addison-Wesley Publishing Company, 1993, pp 217-251

[PN84] Pereira, L and Nasr, R , "Delta Prolog A Distributed Logic Programming Language", Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, Japan, 1984, pp 283-291

[TA81] Tanenbaum, A , "Computer Networks", Prentice Hall, 1981, pp 177-180

[SH83] Shapiro, E "Concurrent Prolog a progress report", IEEE Computer vol 19 no 8, pp 44-58, 1983

VITA

Surname: Rintoul | Given Names: Kevin

Place of Birth: Montreal, Quebec | Date of Birth: December 23, 1964

Educational Institutions Attended

University of Victoria	1983-1988
University of Victoria	1991-1994

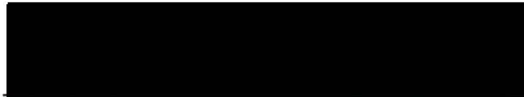
Degrees Awarded

B.Sc.	University of Victoria	1988

Partial Copyright License

I hereby grant the right to lend my thesis (the title which is shown below) to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission

Remote Predicates For Prolog:
A Basis for Declarative Client/Server Applications

Author 

Kevin Rintoul

Date April 14, 1994