Practical Development of Internet Prolog Applications using a Java Front End

Samhaa R. El-Beltagy, Mahmoud Rafea, & Ahmed Rafea

Central Lab for Agricultural Expert Systems
Agricultural Research Center
Ministry of Agriculture and Land Reclamation.
Cairo, Egypt

El-Nour St. Dokki 12311,Giza Egypt

{samhaa, mahmoud, rafea}@esic.claes.sci.eg

Abstract: This paper introduces a general architecture that could be employed to many Prolog applications to make them available on the Internet. The approach presented makes use of client-server architecture where the client is a relatively intelligent front end written in Java, and the server is the Prolog based application. Two applications developed using this architecture: an intelligent image retrieval application, and a toy expert system, are discussed.

Keywords: Prolog, Java, Client-server architectures, Hierarchical Classification, Expert Systems, Internet applications.

Introduction

The widespread use of the Internet and the World Wide Web has motivated much work with the aim of providing interactive applications on the Internet. Sun's *Java*[Java97], Netscape plug-ins [BuPlg97], and Microsoft's *ActiveX*[ActX97] are perhaps the most famous of these endeavors. In order to support this new technology, Internet browsers, and tools have undergone numerous evolutions. These advances have made the Internet a very suitable medium for providing sophisticated services including those of logic based nature. Traditionally, one of the problems faced by such applications, was that the languages in which they are usually developed (logic programming languages) was not directly supported on the Internet. Although, this lack of support has deprived Internet users from using applications that use extensive reasoning facilities which are most naturally supported by these languages, various work arounds that make use of the emerging technology are currently being devised.

The use of Netscape Plug-ins is one of these alternatives. However, through our experience in devising a Prolog plug-in[SiNP97] we have identified several disadvantages associated with such an approach. Plug-ins can only support light weight applications, they are platform specific, and a user must download the plugin and install it before using it. Other approaches propose the use of scripting languages with HTML[3]. While this approach might be effective, it is complex, relies on heavy communication, and cannot be reused without changes to support other applications.

This paper introduces an approach that takes advantage of the current technology through the use of Java, to bring the power of Prolog-based applications to the Internet. The approach adopted follows a client/server architecture where the client is a front end with reasonable intelligent qualities and the server is the Prolog based application. Java has been chosen for the implementation of the front end for a number of reasons which are described briefly in the following points:

- it is platform independent
- it allows the easy implementation of effective GUIs,
- it is a safe language (no pointers, and scripts are tagged)

- it supports a number of network communication mechanisms one of which is socket based communication which is used by the approach presented in this paper
- and most important of all, it is Internet ready (applets are already the de-facto standard for executable content on the Web)

The first part of this paper introduces the application architecture devised to support this work. Then, two applications: an intelligent image retrieval application, and a toy expert system, which have been developed using this approach, are discussed.

Application Architecture

In the approach presented in this paper, the application consists of a mobile client component, a static server component, and two communication components: one for the client and one for the server. The communication between the client and the server is based on TCP/IP sockets where exchange of ASCII based string messages is facilitated. The content of the messages is dependent on an application defined protocol supported by the socket components. Figure 1, depicts the relationship between the different application components.

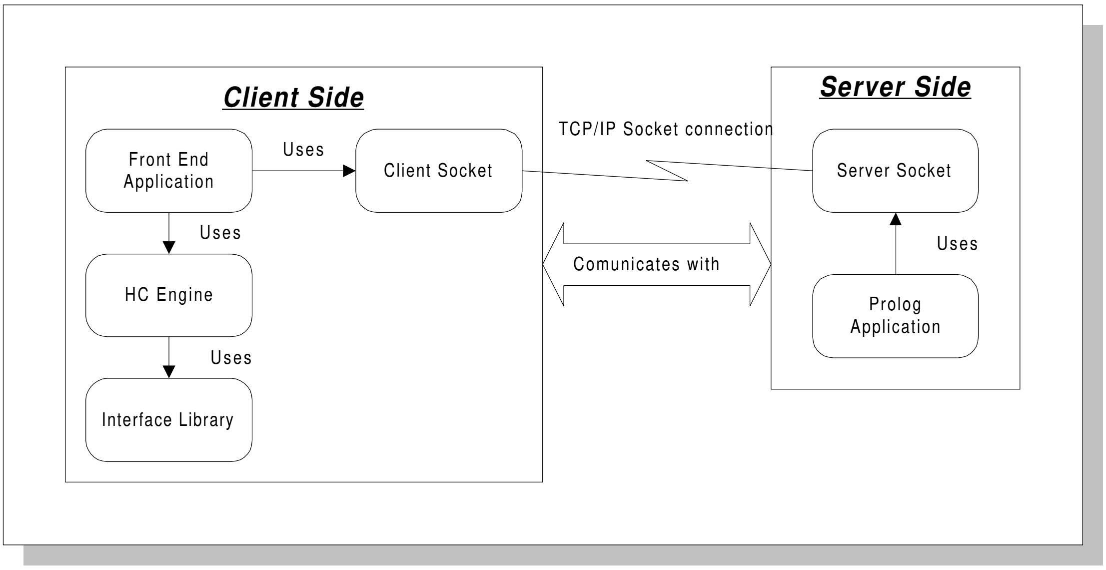


Figure 1: Relationship between the application components

The client side represents the application front end which is entirely written in Java and is comprised of a number of components. At the heart of the client side is the interface communication model which is used for intelligent data collection. The server side, represents the problem-solving/application-code, and the knowledge-base/data components and is entirely written in Prolog.

The underlying assumption behind this work is that many applications could be separated into two parts: an application component and an interface or a front end component. However, this work acknowledges the fact that application separation is not usually a straight forward task since the control of the interface is usually managed by the application itself. For instance, in many expert systems, the questions to be asked are determined by answers to previously asked questions. In this case several solutions are possible. The first and the simplest of these, is devising an application specific interface where the user is presented with all possible inputs. Needless to say, this approach will not meet the needs of any reasonably large application, and in addition, will confuse the user.

Another approach, entails keeping control embedded within the problem solving server application. In that case, the interface will be used to present the user with an input request upon receiving such a request from the server(Szeredi, et al, 1996). While this approach might seem reasonable, it suffers from major limitations. First, it relies on heavy communication between the server application and the client front end, so the user may have to wait for prolonged periods of time depending on the network traffic and band width. In case of synchronous communication, it can engage the server in one connection for an indefinite amount of time, making it impossible for other users to make use of that same server. Although time-out operations could be implemented to avoid indefinite postponement, the application server will still not be fully utilized. In case of asynchronous communication the server will have to maintain extra knowledge such that data inputs could be mapped to application clients. This would be necessary in order to maintain data values that are consistent with its clients.

If however, the interface component employed a communication model that had just enough knowledge about which inputs it should ask about and in which cases, then the client could use this knowledge to collect all needed inputs in an intelligent fashion, then, them in one batch to the server for processing. In this case the connection between the client and the server will only be open for the period of sending the inputs, processing them at the server and receiving the output. For most practical applications, this period is usually reasonably short. Meantime, if other clients need to service a request, the request will be placed in a wait queue where the waiting time will be short enough to make that wait transparent. However, care must be taken in the selection of the queue size. Otherwise, if the number of the clients for the application grows, then wait times might also grow to unsatisfactory figures.

In order to bring this concept into existence, we have implemented an intelligent client side communication model which is described in detail in the next section.

Client side Components

The client side is composed of a communication model and a socket interface. In order to build the communication model, it uses components from implemented reusable libraries.

The Communication Model

The implemented communication model employs a similar, though not identical, strategy to that employed by Hierarchical Classification problem solvers. Hierarchical Classification(HC) is a problem solving method identified by Chandrasekran for solving diagnostic types of problems as part of his Generic Task approach to expert system development. In HC, knowledge is represented as hypotheses hierarchically organized in a tree structure such that general hypothesis are always above more specific ones in the tree. Using a control strategy known as establish and refine, hypothesis are explored top down. If a hypothesis at the top level succeeds (establishes), its immediate descendants are required to establish themselves one by one. This process of attempting to establish the descendants is referred to as "refining" the parent hypothesis. If, on the other hand, a hypothesis fails, then it is said to be ruled out and so are all the hypothesis beneath it in the tree[1].

In our model, all knowledge components for which input is desired, are also organized in a hierarchical fashion. The main difference however, is that no actual problem solving takes place. The evaluation of the tree nodes in this case, only serves to intelligently deduce the next data items for which to ask the user. The process of building this hierarchy is a fairly simple one given that dependencies and relations between data input items, are known.

Re-Usable Libraries

In order to generalize development using this approach, a library that supports hierarchical classification was implemented in Java. In this library a HC engine, as well as node definitions and manipulation operations, were supplied. In order to build specific applications, the user only has to extend the HC class, and define nodes in which variables are object instances declared using another reusable interface library components.

The interface library was built so as to provide a set of standardized dialog components for different data types. This library manages all error handling and presentation details so as to conceal these low level interface aspects from the application builder. Implemented classes, include support for boolean, single selection lists, multiple selection lists, strings, integers, reals, and date variables. The library is generic and extensible. Using this library all a programmer has to do is to declare a variable object using any of the classes defined for each data type. When a value is needed, a simple method getValue() is invoked upon that object. Upon this invocation, a dialog requesting the value of that variable is presented to the user and later returned to the caller.

The client socket component

The client socket in conjunction with the server socket, act as a communication interface between the Java client front end, and the Prolog-based application. The client socket is implemented as a reusable Java Class. Each time the client front end needs to communicate with the Prolog application, it creates an instance of this component giving it both the host name and the port number on which the Prolog application's server socket resides. Since the front end is usually implemented as a Java applet, host information is usually passed to it as parameters from its initiating HTML page for the purpose of generality.

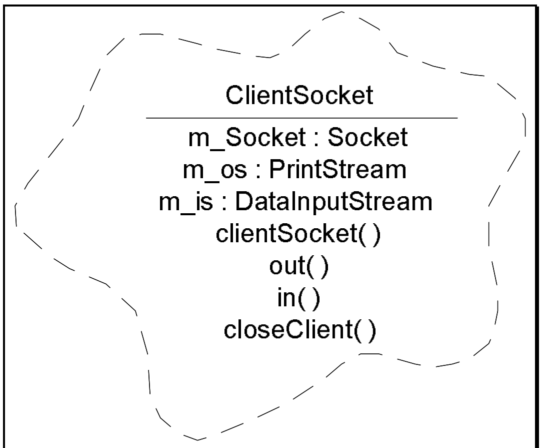


Figure 2: The client socket class

The client socket provides two primary methods to handle I/O between the client and the server. The first of these (**`void out(String outString))`**) is a method for writing a string on the TCP stream connection between the client and the server, while the second (**`String in()`**) is a method for reading from that same stream. Using these two simple methods, sufficient bases for communication between the client and the server, are established.

```
public ClientSocket(String host, int port) {
        try {
                m_Socket = new Socket(host, port);
                m_os =new PrintStream(m_Socket.getOutputStream()); //get output stream
                m_is =new DataInputStream(m_Socket.getInputStream());//get input stream
        }
        catch (UnknownHostException e) {
                System.err.println("Don't know about host: " + host);
        }
        catch (IOException e) {
                System.err.println("Couldn't get I/O for the connection to: " + host);
        }
}
```

Figure 3: Code responsible for socket creation and initiation

Server Side Components

The server side consists of an application component which in turn, could be composed of a number of components, and a server socket. .

The server socket component

This component is simple. It is linked with the server application code. The server application is responsible for its initialization. Its functions are:

- Creating a socket and binding it to a specific port number.
- Creating a backlog queue of a particular length.
- Opening and closing a client stream.
- Reading from and writing to the client stream.
- Managing the clients in the queue sequentially on the basis of first come first served.

The service is achieved by passing Prolog terms sent by the client to the server application through a call to the procedure **`application_handle/2`**. Consequently, the server application must define the procedure **`application_handle/2`** so that it never fails and returns with the term that will be sent to the client. The server socket component code is shown in figure 4.

It should be remarked that the success of this architecture depends on the efficiency of the server application performance. The shorter the time needed to process the client request, the better the behavior of the application on the Internet.

```
start_server(Port, QueueLength) :-
        socket('AF_INET', Socket),
        socket_bind(Socket, 'AF_INET'(localhost, Port)),
        socket_listen(Socket, QueueLength),
        loop(Socket).

loop(Socket) :-
        socket_accept(Socket, Client, Stream),
        on_exception(RE, get_request(Stream, Request), h1(Stream,Client,RE)),
        on_exception(PE, process(Request, Stream, Client), h1(Stream, Client, PE)),
        loop(Socket).

get_request(Stream, Request):-
        read(Stream, Request).

process(bye, Stream, _Client) :-
        close(Stream).
process(Request, Stream, Client) :-
        on_exception(E, user:application_handle(Request,Result), h2(Stream, no, E)),
        return(Stream,Result),
        on_exception(RE, get_request(Stream, Request1), h1(Stream,Client,RE)),
        on_exception(PE, process(Request1, Stream, Client), h1(Stream, Client, PE)).

return(Stream,Result) :-
        nonvar(Result), !,
        format(Stream, "~w~n", [Result]),
        flush_output(Stream).
return(_, _).

                                   h1(Stream, C, E) :-
        format(Stream, "error~n",[]),
        format(user_output, "Client: ~w~nError: ~w",[C, E]).

h2(Stream, Ack, E) :-
        format(Stream, "~w~n",[Ack]),
                       format(user_output, "Application Error: ~w~n",[E]).
```

Figure 4: Prolog code of the server socket component

Image search application

The objective behind the development of this application is two-fold. The first is to aid a remote Internet client in searching for the images that best matches his input observations[CuIS97]. The second is to cooperate with an expert system explanation agent[1]. As described by our general architecture, the application is composed of two parts: a static server component and a mobile client component. Figure 5 shows a simplified diagram of interactions between the application components.

The static server component

The static server application was implemented using SICStus Prolog[2]. It, consists of a reusable search engine, and a domain specific image database mapping observations to images (figure5). The input to the search engine is a list of observations and the output is a list of image-file structures. Each image-file structure consists of a file name and the actual observations associated with the image in the file. The database is constructed though the use of two predicates: finding/4 and image/2.

The database predicates take the following pattern:

finding(attribute, value, op, key).
image(key, file)

The input list defined by the application protocol takes the following form:

[attribute1 op value1, attribute2 op value2, ...]

---

[1] The explanation agent will employ a similar architecture to that discussed in this article. It is now under development.

where <op> is among = , > , < , >= , or =<.
The output is a string of the following form:

"f:<filename1>,<attribute><op><value>,..., f:<filename2>....."

OR

"e:<Error>"

where <attribute><op><value>, ... are those belonging to the image.

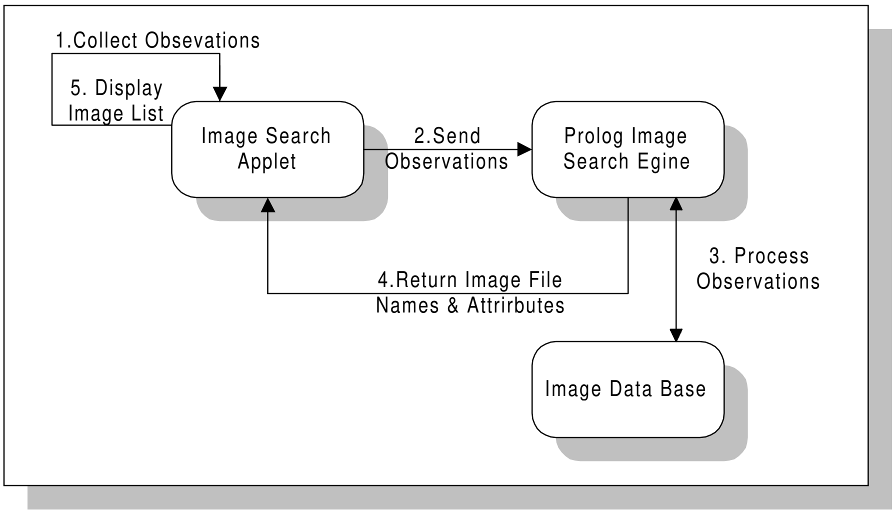


Figure 5: A simplified diagram of the image fetching process.

```
get_image(List, Output) :-
        check_syntax(List), !,
        findall(File, (list_to_key(List, Fs), image_file(Fs, File)), Files1),
        (       Files1 = [] ->
                (       list_to_key(List, Fs) ->
                        length(Fs, Len),                        % Collect all subsets
                        findall(I-SubL,(sublist(SubL, Fs), length(SubL, SLen),
                                SLen > 0, SLen < Len, I is 1000 // SLen), SubLs),
                        (       SubLs = [] ->  % Singlton and fail to get an image
                                Output = 'e:No image file found'
                        ;       keysort(SubLs, SubLs1),                 % try maximum
                                SubLs1 = [I-SubFs|Rest],                % get first subset
                                best_image_file(Rest,I,SubFs,[], Files2),
                                (       Files2 = [] -> Output = 'e:No image file found'
                                ;       format_output(Files2, Output)
                                )
                        )
                ;Output='e:Some input observation(s) is/are not defined'
                )
        ;       format_output(Files1, Output)
        ).
get_image(_List, O) :-
        O='e:The input is not a valid list ([a op v,...] or op is not defined.)'.


best_image_file([], _, SubFs, Files1, Files) :- % Handle input of two observations
        findall(File, image_file(SubFs, File), Files2),
        (Files2 \== [] -> append(Files2, Files1, Files) ; Files = Files1).
best_image_file([I-Fs|Rest], I1, SubFs, Files1, Files) :- % Collect same weight images
        findall(File, image_file(SubFs, File), Files2), Files2 \== [], !,
        append(Files2, Files1, Files3),
        (I = I1 -> best_image_file(Rest, I, Fs, Files3, Files) ; Files = Files3).
best_image_file([I-Fs|Rest], I1, _, Files1, Files) :-           % Keep on looping
        (       Files1 = [] -> best_image_file(Rest, I, Fs, Files1, Files)
        ;       I = I1      -> best_image_file(Rest, I, Fs, Files1, Files)
        ;       Files = Files1
        ).

. . . .
. . . .
. . . .
```

Figure 6: The Core of Prolog Image search engine

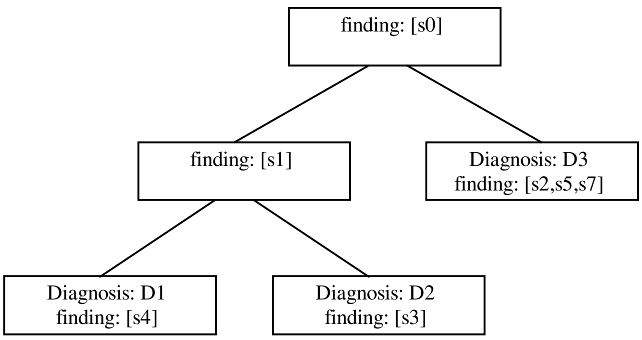
The client component

The client component is composed of an applet which includes a set of observations, a HC component where the observations are distributed over nodes, a parser, and a simple GUI. Upon opening the page that contains the applet, the applet is downloaded and the control interface appears. Using this GUI, a user may initiate an image search. A series of question dialogs then start to appear. The ordering of the appearance of the dialogs is controlled by the answers of the user to the previously presented questions, through the HC component. This process, constitutes the data collection phase. After this phase is completed, a string representing the aggregate of the inputs is prepared and passed to a new instance of the client socket which establishes a link with the server and uses it to send the string to the Prolog search engine residing on the server. After the search engine has processed the input, the client socket reads the output which represents the file names with observations that best match the user's inputs, and the exact observations associated with each, and passes it to the applet. The client socket then terminates its connection with the server and dies. The client processes the output through the parser, and then presents a list of all image names to the user in a list. Upon selecting an image to view, the client fetches the file using the URL passed to it by the server, and uses the observations requested by the user and the actual observations associated with the image to determine matching and mismatching features. The image is then presented to the user along with the match information.

Expert system application

This application is a toy expert system for diagnosing car faults[CaES97]. The server application is built in SICStus Prolog[2]. It consists of the application knowledge base and a library of general problem solving methods[2]. The details of the problem solving library are out of the scope of this article. Actual expert systems that have already been developed at the Central Laboratory for Agricultural Expert Systems, are currently being ported to the proposed architecture.

In the car fault application, the knowledge base consists of nodes for a hierarchical classification generic task and a global database object. The latter represents a blackboard which is used by the reasoning task. The nodes are depicted in figure 7.




```
D1 = Condensation in distributor cap
D2 = Faulty distribution arm
D3 = Worn distribution brushes
s0 = problem(yes)).
s1 = starter(yes)).
s2 = starter(no)).
s3 = cuts_at_speed(yes)).
s4 = cuts_at_speed(no)).
s5 = start_diff(yes)).
s6 = start_diff(no)).
s7 = cuts_shortly(yes)).
s8 = cuts_shortly(no)).
```

Figure 7: Nodes of HC for the car fault expert system

The input the to the expert system is a set of observations and the output is the diagnosis and advice. Inputs to the expert system are collected via client applet and sent to the server application in a similar fashion to that of the image application. However, the applet GUI is different than that of the image search applet in that it provides a text box where the diagnosis and advice of the expert system appear.

---

[2] This library is under publication.

Conclusion and future work

Through the architecture introduced in this paper, many applications developed in Prolog could be easily ported to the Internet. The Prolog socket component and the Java libraries are reusable without any modification. An application builder only needs to define relations between data input items using the communication model presented, and adapt the Prolog application for batch interaction with the client component.

The implementation of work presented here could be greatly simplified through development automation. For that purpose, a number of tools have been implemented to support this work and its consistency across the client and server components. However, discussion of these tools, is beyond the scope of this paper.

Future work will focus on enhancing the image tool by using a static databases to replace the Prolog database. It will also implement protocol standardization through support of KQML based message exchange. A number of existing Prolog based expert systems are currently being ported to this architecture.

References

1. El-Beltagy, S., Rafea, A. Kamel, A., Sticklen, J., Schulthess, U., Ward, R. (1995). "An Expert System For Wheat Disorders Diagnosis And Treatment Using A Hierarchical Classification Problem Solver". 2ND IFAC/IFIP/EurAgEng Workshop on Artificial Intelligence in Agriculture, Wageningen, Netherlands, Pergamon.
2. SICS Programming Systems Group. SICStus Prolog User's Manual. Swedish Institute of Computer Science, June 1995.
3. Szeredi, P., Molnár, K., and Scott R. "Serving Multiple HTML Clients from a Prolog Application", Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications , Bonn, Germany, 1996.

Internet References

[ActX97] Activex.com
URL: http://www.activex.com/

[BuPlg97] Building a Plug-in from the ground up
URL: http://home.netscape.com/misc/developer/conference/proceedings/ci5/index.html

[CaES97] Car Fault Expert System
URL: http://tomato.claes.sci.eg/carFault/

[CuIS97] Cuptex Image Server
URL: http://tomato.claes.sci.eg/cuptexImgs/

[Java97] The Java home page.
URL: http://java.sun.com

[SiNP97] SICStus Prolog Netscape Plugin
URL: http://potato.claes.sci.eg/claes/plugin/npsp.html