# Brief Overview of Different Prolog Implementations Libraries and tuProlog Socket Communication

Practical Project of Computational Models and Languages LM

Author: Mirco Bordoni

Teacher: Mirko Viroli

Academic Year: 2011 - 2012

# Outline

- Main libraries implemented in four different Prolog implementations: GNU Prolog, SWI-Prolog, SICStus Prolog and Ciao Prolog.
- Analysis of Socket library in the previous Prolog implementations
- Socket library analysis and implementation for tuProlog
- Multi-threading issues
- Conclusions

# Main Libraries in other Prologs

- Constraint Logic Programming over Finite Domains, Rationals and Reals
- Operative System Interface
- Interface to other languages (C, C++, Java, .NET)
- Sockets I/O
- Multi-Processing and Multi-Threading
- Data Structures Definition (Lists, Key-Value Lists, Trees, Graphs, Sets)
- Data Structures Navigation
- ODBC Interface
- Semantic Web (RDF, OWL, XML, etc.)
- Debugging

# Socket I/O in GNU Prolog

- *socket(+Domain, -SocketId)*: Socket creation
- *socket_close(+SocketId)*: Close an opened Socket
- *socket_bind(+SocketId, +Socket_Address)*: binds the socket identied by SocketId to a specic address
- *socket_connect(+SocketId, +Socket_Address, -StreamIn, -StreamOut)*: connect a Socket to a specified address
- *socket_listen(+SocketId, +Length)*: set the maximum number of pending connections
- *socket_accept(+SocketId, -Client, -StreamIn, -StreamOut)*: accept a connection to *SocketId* obtaining the address of the connecting client, a stream for input and another for output. It is a blocking predicate

Multi-processing support to cope with blocking predicates using forks and pipes.

# Socket I/O in SWI-Prolog

- *tcp_socket(-SocketId)*: create a new socket identied by SocketId
- *tcp_close_socket(+SocketId)*: close an opened socket identied by SocketId
- *tcp_open_socket(+SocketId, -InStream, -OutStream)*: get Input and Output Streams to communicate through the socket
- *tcp_bind(+Socket, ?Port)*: bind the socket to a specic port on the current machine
- *tcp_listen(+Socket, +Backlog)*: set the number of maximum pending connections
- *tcp_accept(+Socket, -Slave, -Peer)*: wait for a client connection to the socket. On success, it creates a new socket for the client and binds the identifier to Slave. Peer is bound to the IP-address of the client.
- *tcp_connect(+Socket, +Host:+Port, -Read, -Write)*: connect a client to a specific *host:port*
- *tcp_connect(+Socket, +Host:+Port)*: same a the previous one, but to use the socket we have to call *tcp_open_socket/3*

Multi-threading support to cope with blocking predicates!

# Server Example in SWI-Prolog

```
create_server(Port):- tcp_socket(Socket),      %Opening Socket
                      tcp_bind(Socket, Port), %Bind it to a Port
                     tcp_listen(Socket, 5),   %Maximum pending connections
                                              %set to 5
                     tcp_open_socket(Socket, AcceptFd, _),%Opening an Input
         stream
                     <dispatch>                       %user-defined behaviour
dispatch(AcceptFd) :- tcp_accept(AcceptFd, Socket, _Peer),    %wait for an
                                                        %incoming connection
                     thread_create(process_client(Socket, Peer),_,
                                [detached(true)]), %Create Thread
                     dispatch(AcceptFd). %back to waiting for a connection
process_client(Socket, Peer) :- % Thread Behaviour
                      setup_call_cleanup(tcp_open_socket(Socket, In, Out),
                      handle_service(In, Out),
                      close_connection(In, Out)).
close_connection(In, Out) :- % Close streams
                      close(In, [force(true)]),
                      close(Out, [force(true)]).
```

# Client Example in SWI-Prolog

```
create_client(Host, Port) :- setup_call_catcher_cleanup(tcp_socket(Socket),
                                 tcp_connect(Socket, Host:Port),
                                 exception(_),
                                 tcp_close_socket(Socket)),
                             setup_call_cleanup(tcp_open_socket(Socket, In, Out),
                             chat_to_server(In, Out),
                             close_connection(In, Out)).

close_connection(In, Out) :- close(In, [force(true)]),
                             close(Out, [force(true)]).

chat_to_server(In, Out) :-
     ...
```

# Socket I/O in SICStus Prolog

- *socket_client_open(+Addr, -Stream, +Options)*: open a new client socket and try to connect to a specied address
- *socket_server_open(?Addr, -ServerSocket, +Options)*: create a Server Socket and bind it to the specic address
- *socket_server_accept(+ServerSocket, -Client, -Stream, +StreamOptions)*: waits for an incoming connection to *ServerSocket*
- *socket_server_close(+ServerSocket)*: close *ServerSocket*
- *socket_select(+ServerSockets,-SReady, +ReadStreams,-RReady, +WriteStreams,-WReady, +Timeout)*: check for server sockets with incoming connections, streams on ReadStreams ready for input, and streams on WriteStreams ready for output. The ready server sockets are returned (in the same order) in SReady, the ready input streams in RReady, and the ready output streams in WReady.

No support to Mutli-threading. The only way to avoid useless waitings is the predicate *socket_select/7.*

# Socket Library for tuProlog

It is possible to think about a Socket Library for tuProlog because:

- tuProlog is written in Java
- it is easy to write a new library in Java for tuProlog
- Java supports TCP sockets
- Java supports Multi-threading

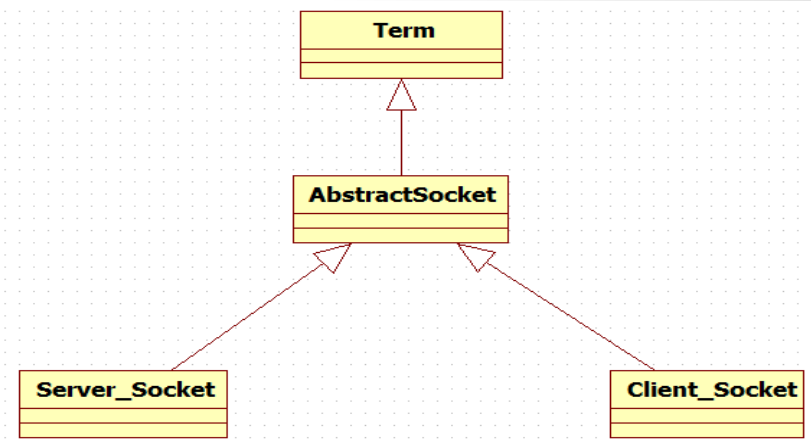
# Socket Library for tuProlog - Interface

Socket library has to implement the Interface:

```
public interface ISocketLib {
    public boolean tcp_socket_client_open_2(Struct Address, Term
                   Socket);
    public boolean tcp_socket_server_open_3(Struct Address, Term
                   Socket, Struct Options);
    public boolean tcp_socket_server_accept_3(Term ServerSock,
                   Term Client_Addr, Term Client_Slave_Socket);
    public boolean tcp_socket_server_close_1(Term serverSocket);
    public boolean read_from_socket_3(Term Socket, Term Msg,
                   Struct Options);
    public boolean write_to_socket_2(Term Socket, Term Msg);
    public boolean aread_from_socket_2(Term Socket, Struct
                   Options);
}
```

# Socket Library for tuProlog - New Typed Terms

To work with sockets it might be useful to implement new typed Terms. I decided to add three new Terms as shown in the picture.




AbstractSocket extends Term and overrides its main methods.
It then declares four new abstract methods:

- `public abstract boolean isClientSocket();`
- `public abstract boolean isServerSocket();`
- `public abstract Object getSocket();`
- `public abstract InetAddress getAddress();`

These four abstract operations will be implemented in *Server_Socket* and *Client_Socket*.

# Socket Library for tuProlog - Server Side

The server side main functionalities are:

- `tcp_socket_server_open(+Address, -Socket, +Options)`: open a new Server Socket bound to *Address. Socket* becomes bound to a *Server_Socket* term. *Address* has to be in the form IP:Port. *Options* can only contain the term *backlog(n)* to set the maximum number of pending connections.

- `tcp_socket_server_accept(+ServerSock, -Client_Addr, -Client_Slave_Socket)`: this is a blocking predicate and when it succeeds it unifies Client Addr with the client address in the form Address: Port and Client Slave Socket with the new socket created to communicate with that client.

- `tcp_socket_server_close(+ServerSocket)`: close a Server_Socket

# Socket Library for tuProlog - Client Side

On the client side a user can open a socket that automatically connect to the specied address, where a server socket should be listening.

The predicate used to open a new Client Socket is:
`tcp_socket_client_open(+Address, -Socket)`

The argument Address has to be in the form *Address:Port* and it is the address of the server listening for connection requests.
If the connection succeeds the variable *Socket* is unified with the new *Client_Socket* created.

# Socket Library for tuProlog - Communication

Once a connection is established two sockets can communicate using the following predicates:

- `write_to_socket(+Socket, +Msg)`: write a message in the OutputStream associated to the Socket passed (it has to be a Client_Socket)
- `read_from_socket(+Socket, -Msg, +Options)`: read a message from a Socket that should be bound to a *Client_Socket*. If it succeeds the received message is unified with variable *Msg*. This is a blocking predicate, but the user can specify a timeout. If nothing is read until the timeout expires, the predicate fails. To set a timeout an option *timeout(millis)* has to be passed through the list *Options*
- `aread_from_socket(+Socket, +Options)`: asynchronous read from the specified *Client_Socket*. When something is read it is asserted in the Prolog theory using the method *assertA*. The user can specify two options for this predicate: *timeout(millis)* and *assertZ*.

# Thread Safety in Theory Management

The predicate *aread_from_socket*/3 uses the methods *assertA* and *assertZ* of the class *TheoryManager*.

To grant that no race conditions happen, every method of the class *TheoryManager* has to be synchronized, so it can provide mutual exclusion in theory access.

The access to the theory is always done using *TheoryManager* methods!

Clauses are retreived from the *ClauseDatabase* using the method *find* of the class *TheoryManager*!

# Basic Multi-Threading library

During the implementation of the Socket Library I implemented a basic support to multi-threading using a new engine to solve a goal passed by the user.

The only predicate I have implemented is:

```
create_thread(+Goal, -Id, +Options)
```

where *Goal* is the goal to solve. The theory of the new engine is set to the theory get from the engine that calls the predicate *create_thread*.

By now it is not possible to use the same engine to solve different goals concurrently because the method *solve* is declared as synchronized.

A deeper analysis of thread-safety is needed to achieve a full support to multi-threading.

# Changes in tuProlog source code (1/4)

The method *unify* of the class *Var* has been changed. In this method there is a check on the type of the Term passed for the unification process. If the term is an AbstractSocket it has to succeed, so the block:

```
}else if (!(t instanceof Number)) {
    return false;
}
```

has been changed into:

```
}else if (!(t instanceof Number) && !(t instanceof
AbstractSocket)) {
    return false;
}
```

# Changes in tuProlog source code (2/4)

When a user stops the engine while solving a goal, all threads have to be stopped and all sockets have to be closed.

For this reason I have added to the class *Library* a method called *onSolveHalt()*.This method is then overridden in the library *SocketLib* and it simply calls *onSolveEnd()* to close all sockets and stop all threads.

I have also added to the class *LibraryManager* the method *onSolveHalt()* that calls the method *onSolveHalt()* of each library.

So the method *solveHalt()* in the EngineManager becomes:

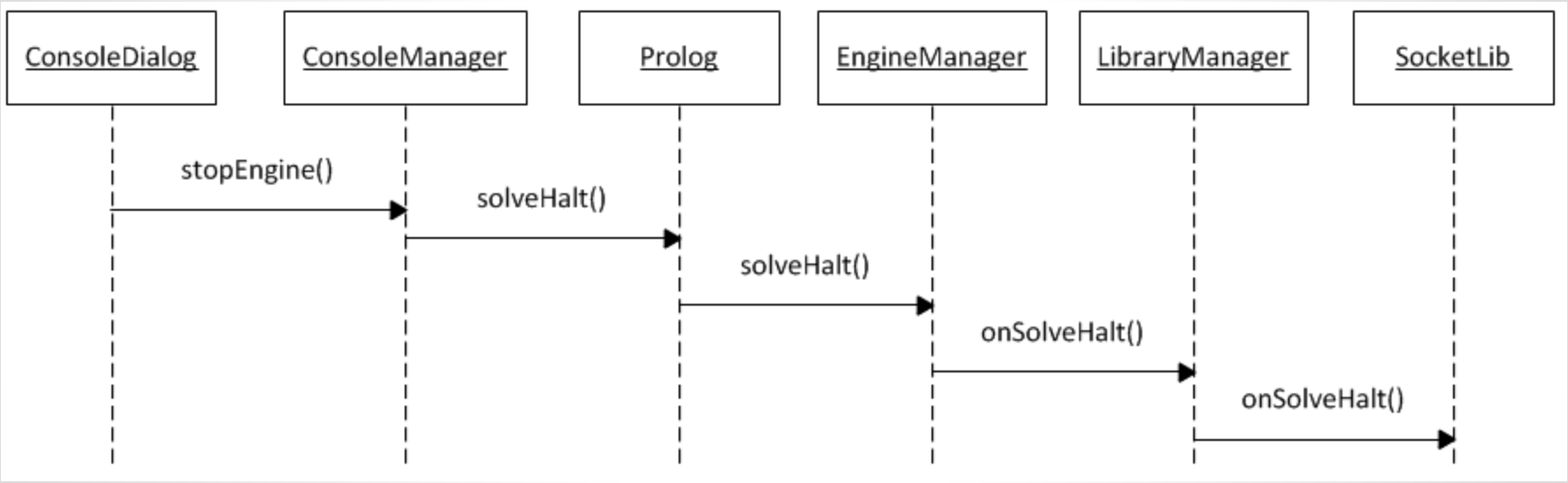
```
public void solveHalt() {
    env.mustStop();
    libraryManager.onSolveHalt();
}
```

while before it was:

```
public void solveHalt() {
    env.mustStop();
}
```

# Changes in tuProlog source code (3/4)

This is what happens now when the button *Stop* is pressed by the user:



# Changes in tuProlog source code (4/4)

To support basic multi-threading a change to the method *getTheory* of the class *Prolog* has to be done.

Before it was:

```
public synchronized Theory getTheory() {
    try { return new Theory(theoryManager.getTheory(true));
    } catch (Exception ex){return null;}
}
```

Now it is:

```
public Theory getTheory() {
    try { return new Theory(theoryManager.getTheory(true));
    } catch (Exception ex){return null;}
}
```

If the method remains synchronized, new threads can't read the theory until Prolog engine has completely solved the goal. Thread safety is assured because *theoryManager.getTheory(..)* is Thread Safe!

# Example 1 - Client-Server echo

In this example Client sends terms to a Server that sends them back adding them to a predicate *echo*. Client code is:

```
client(X,Y,Z):-tcp_socket_client_open('127.0.0.1:4444',Sock),
    write_to_socket(Sock,test1), read_from_socket(Sock,X,[]),
    write_to_socket(Sock,test2), read_from_socket(Sock,Y,[]),
    write_to_socket(Sock,test3), read_from_socket(Sock,Z,[]).
```

While the server code is:

```
server(X,Y,Z):- tcp_socket_server_open('127.0.0.1:4444', Sock,[]),
                tcp_socket_server_accept(Sock,ClientAddr,Slave),
                read_from_socket(Slave,X,[]),
                write_to_socket(Slave,echo(X)),
                read_from_socket(Slave,Y,[]),
                write_to_socket(Slave,echo(Y)),
                read_from_socket(Slave,Z,[]),
                write_to_socket(Slave,echo(Z)),
                tcp_socket_server_close(Sock).
```

# Example 2 - Superserver

```
start:- tcp_socket_server_open('127.0.0.1:4444', Sock,[]),superserver
(Sock).
superserver(Sock):- tcp_socket_server_accept(Sock,ClientAddr,Slave),
          create_thread(server(ClientAddr,Slave),_,[]),superserver(Sock).

server(ClientAddr,Slave):- read_from_socket(Slave,Msg,[]),
                           factorial(Msg,Res),
                           write_to_socket(Slave,Res),
                           checkNewReq(Slave).

checkNewReq(Slave):- end,retract(end),!.
checkNewReq(Slave):- newReq(N), retract(newReq(N)), factorial(N,Res),
                     write_to_socket(Slave,Res),
                     aread_from_socket(Slave,[]),
                     checkNewReq(Slave).
checkNewReq(Slave):- aread_from_socket(Slave,[]),
                     checkNewReq(Slave).
```

Factorial calculation is omitted.

# Example 2 - Client for Superserver

```
client(Msg,Msg1):- tcp_socket_client_open('127.0.0.1:4444',Sock),
                   write_to_socket(Sock,6),
                   read_from_socket(Sock,Msg,[]),
                   write_to_socket(Sock,newReq(4)),
                   read_from_socket(Sock,Msg1,[]),
                   write_to_socket(Sock,end).
```

# Future Work

Some future work that can be done on this library can be:

- Implementation of UDP socket communication.
- Extend Multi-Threading Support. For example it may be interesting to develop a multi-threading library that runs different threads sharing the same engine or to react to some kind of events using different threads. By now it is not even possible to see the output of a thread.