Appendix A
How to implement a Prolog node

> Vision without execution is just hallucination.
>
> *Thomas Alva Edison*

We would have loved to be able to present a stable, speedy and secure implementation of a Prolog node, ready to be deployed to help building the Prolog Web. However, there exists no such implementation at this point in time. There are some proof-of-concept implementations, but they are neither stable nor speedy, nor secure. How can we build one that is? And how can we build *more* than one, so that we can make sure that interoperability across different implementations works as intended?

A.1 Wrapping a node around an existing Prolog system

> Make it work, then make it beautiful, then if you really, really have to, make it fast. 90% of the time, if you make it beautiful, it will already be fast. So really, just make it beautiful!
>
> *Joe Armstrong*

It is likely that the first implementations of Prolog nodes would be Prolog systems providing as libraries whatever is required to comply with Web Prolog requirements. This is how our proof-of-concept implementations were built.

Some really excellent Prolog systems exist out there, so if you are a Prolog implementor, one obvious advantage with this approach is that most of the necessary work has already been done. The amount of additional work required to implement a node depends on which system it is built on top of.

In this section, we look at different ways to wrap a node around a system that supports the ISO Prolog working draft for threads.[1] We are aiming for an almost complete ISOBASE node, as well as an ACTOR node, albeit less complete.

Using SWI-Prolog, we show a way to implement the stateless HTTP API. We do not focus solely on semantics, but on performance too. In particular, we devise a way to optimize the API by avoiding the spurious recomputation of solutions that a naive implementation would have to do. Furthermore, we implement a version of `rpc/2-3` on top of the stateless HTTP API.

We also implement specifications for how we believe predicates such as `spawn/2-3`, `!/2` and `receive/1-2` should work. On top of actors, we implement the behavior of Prolog toplevels. These implementations focus on semantics rather than performance.

We are seeking, if not beauty, then at least as much clarity and simplicity as possible. Our implementations are only partial, but we also indicate what else would be needed to complete them.

A.1.1 Implementing an ISOBASE node

A Prolog ISOBASE node is equipped with a stateless HTTP API. Managing this API is actually the only task its node controller is resonsible for. It means that we can make good use of a library for building web servers. Here is how a web server may be written in SWI-Prolog using `library(http/http_server)`:

```
:- use_module(library(http/http_server)).

:- http_handler(root(call), node_controller_isobase, []).

node_controller_isobase(Request) :-
    http_parameters(Request, [
        goal(GoalAtom, [atom]),
        template(TemplateAtom, [default(GoalAtom)]),
        offset(Offset, [integer, default(0)]),
        limit(Limit, [integer, default(10 000 000 000)]),
        format(Format, [atom, default(json)])
    ]),
    atomic_list_concat([GoalAtom,+,TemplateAtom], QTAtom),
    read_term_from_atom(QTAtom, Goal+Template, []),
    compute_answer(Goal, Template, Offset, Limit, Answer),
    respond_with_answer(Format, Answer).

node(Port) :-
    http_server(http_dispatch, [port(Port)]).
```

[1] http://logtalk.org/plstd/threads.pdf

The call to `compute_answer/5` is responsible for the real work here. It takes a goal, a template, an offset and a limit, and computes an answer term serving as a response to the request which can be sent back to the client formatted as Prolog or JSON. There is more than one way to implement this predicate. Let us first look at a simple (but from a performance point of view naive) way of doing it.

SWI-Prolog offers a library predicate `findnsols/4` which provides a useful foundation for our implementation. It is somewhat similar to the standard `findall/3`, but expects an integer `Limit` in its first argument and will generate at most that many solutions. It is also non-deterministic, so on backtracking it will do it again. We borrow an example of its use from the SWI-Prolog manual:[2]

```
?- findnsols(5, I, between(1, 12, I), L).
L = [1, 2, 3, 4, 5] ;
L = [6, 7, 8, 9, 10] ;
L = [11, 12].
?-
```

Another SWI-Prolog library predicate `offset/2` will also prove useful.[3] Its purpose is to *skip* the first *n* solutions to a goal, i.e. the first *n* solutions are computed, but not collected. Here is an example of its use:

```
?- offset(10, between(1, 12, I)).
I = 11 ;
I = 12.
?-
```

Combining `findnsols/4` with `offset/2` allows us to implement a predicate `slice/5` capable of computing a *slice* of solutions to a goal:

```
slice(Goal, Template, Offset, Limit, Slice) :-
   findnsols(Limit, Template, offset(Offset, Goal), Slice).
```

However, we are looking for *answers*, rather than just slices of solutions. By wrapping a call to `slice/5` in a call to `call_cleanup/2` wrapped by a call to `catch/3` we arrive at a predicate `answer/5` capable of producing the four different forms of answer terms that we need:

```
answer(Goal, Template, Offset, Limit, Answer) :-
   catch(
      call_cleanup(slice(Goal, Template, Offset, Limit, Slice),
                   Det = true),
         Error, true),
   (   Slice == []
   ->  Answer = failure
   ;   nonvar(Error)
```

[2] https://www.swi-prolog.org/pldoc/doc_for?object=findnsols/4

[3] https://www.swi-prolog.org/pldoc/doc_for?object=offset/2

```
    ->  Answer = error(Error)
    ;   var(Det)
    ->  Answer = success(Slice, true)
    ;   Det = true
    ->  Answer = success(Slice, false)
    ).
```

This predicate will turn out to be useful in more than one way. In this context it will be used for the implementation of `compute_answer/5`. In this role we want `compute_answer/5` to be deterministic, so since the call to `answer/5` is non-deterministic we need to wrap it in a call to `once/1` like so:

```
compute_answer(Goal, Template, Offset, Limit, Answer) :-
    once(answer(Goal, Template, Offset, Limit, Answer)).
```

The implementation of our simple but naive stateless HTTP API is almost complete, and assuming we also have a suitable implementation of `respond_with_answer/2`, we can now start running a node:

```
?- node(3010).
% Started server at http://localhost:3010/
true.
?-
```

At this point we may want to take the node's stateless HTTP API for a trial run by entering the following URI in a web browser:

```
http://localhost:3010/call?goal=member(X,[a,b])&format=prolog
```

In the browser's window, we should then see the following:

```
success([member(a,[a,b]),member(b,[a,b])],false)
```

By appending `&template=X&offset=0&limit=1` to the URI we should get

```
success([a],true)
```

and by incrementing the `offset` parameter by 1 we should see

```
success([b],false)
```

Note that it is important that we do not expose the node to the whole world at this point, as it is not secure.

A.1.2 Implementing rpc/2-3 on top of the stateless HTTP API

As soon as we have an implementation of the stateless HTTP API, we can easily, by means of two other libraries provided by SWI-Prolog,[4] implement `rpc/2-3` on top of it. Here is the source code:

```
:- use_module(library(http/http_open)).
:- use_module(library(url)).

rpc(URI, Goal) :-
    rpc(URI, Goal, []).

rpc(URI, Goal, Options) :-
    parse_url(URI, Ps),
    term_variables(Goal, Vars),
    Template =.. [v|Vars],
    format(atom(GA), "(~p)", [Goal]),
    format(atom(TA), "(~p)", [Template]),
    option(limit(L), Options, 10 000 000 000),
    rpc_7(Template, 0, L, GA, TA, Ps, Options).

rpc_7(Template, O, L, GA, TA, Ps, Os) :-
    parse_url(ExpandedURI, [
        path('/call'),
        search([goal=GA, template=TA, offset=O,
                limit=L, format=prolog])
      | Ps
    ]),
    setup_call_cleanup(
        http_open(ExpandedURI, Stream, Os),
        read(Stream, Answer),
        close(Stream)),
    rpc_8(Answer, Template, O, L, GA, TA, Ps, Os).

rpc_8(success(Slice, true), Template, O, L, GA, TA, Ps, Os) :-  !,
    (   member(Template, Slice)
    ;   NewO is O + L,
        rpc_7(Template, NewO, L, GA, TA, Ps, Os)
    ).
rpc_8(success(Slice, false), Template, _, _, _, _, _, _) :-
    member(Template, Slice).
rpc_8(failure, _, _, _, _, _, _, _) :- fail.
rpc_8(error(E), _, _, _, _, _, _, _) :- throw(E).
```

[4] See https://www.swi-prolog.org/pldoc/man?section=httpopen and https://www.swi-prolog.org/pldoc/man?section=url

The idea behind this code is to use `http_open/3` in a loop in order to make one or more requests for consecutive slices of solutions to the goal in the first argument using the stateless HTTP API. The URI of each request takes the form

```
BaseURI/call?goal=G&template=T&offset=O&limit=L&format=prolog
```

where `O` is initially `0` and is incremented by `L` between requests.

The most interesting parts of the implementation are the use of the disjunction in the body of the first `rpc/7` clause and the use of `member/2` in the first and second clauses. They are responsible for turning the responses to the deterministic requests made by `http_open/3` into the non-deterministic behavior we want `rpc/2-3` to show.

Let us test our implementation by running an example from Chapter 4, showing how `rpc/2-3` can be used:

```
?- [user].
|: human(plato).
|: human(aristotle).
|: ^D% user://1 compiled 0.00 sec, 2 clauses
true.
?- rpc('http://localhost:3010', human(Who)).
Who = plato ;
Who = aristotle.
?-
```

Note that although the query has two solutions, only one network roundtrip is made, triggered by the following HTTP request:

```
GET http://localhost:3010/call?goal=human(Who)&format=prolog
```

The response contains the following answer term:

```
success([human(plato),human(aristotle)],false)
```

The above code is just a sketch that leaves out some of the details that are necessary for a fully working node. In particular, it does not implement `respond_with_answer/2` and it does not handle syntax errors in queries. None of this would be difficult to add, and with such additions, this section together with the previous one implements the stateless API of an ISOBASE node, as well as the `rpc/2-3` predicate.

A.1.3 Fixing a problem due to spurious recomputation

The above implementation of the HTTP API suffers from a performance problem. The problem is easy to spot when timing a goal simulating a situation where a first solutions takes a long time to compute while a second solution takes almost no time at all – a goal such as the disjunction `(sleep(1), X=foo ; X=bar)` for example. Here is how this looks in a system such as SWI-Prolog:

```
?- time((sleep(1), X=foo ; X=bar)).
% 1 inferences, 0.000 CPU in 1.005 seconds
X = foo ;
% 7 inferences, 0.000 CPU in 0.000 seconds
X = bar.
?-
```

As expected, solving the first disjunct took one second, while the second disjunct took almost no time at all. However, when calling this goal using `rpc/3` with the `limit` options set to `1`, we see the following:

```
?- _URI = 'http://localhost:3010',
   time(rpc(_URI, (sleep(1), X=foo ; X=bar), [limit(1)])).
% 1,984 inferences, 0.001 CPU in 1.006 seconds
X = foo ;
% 1,804 inferences, 0.001 CPU in 1.009 seconds
X = bar.
?-
```

The cause of this problem lies not in the implementation of `rpc/2-3`, but in the HTTP API, and more precisely in the way `compute_answer/5` works. Consider the following call, where the third argument (for the offset) is `1`:

```
?- _Goal = (sleep(1), X=foo ; X=bar),
   time(compute_answer(_Goal, X, 1, 1, Answer)).
% 30 inferences, 0.000 CPU in 1.005 seconds
Answer = success([bar], false).
?-
```

In general, computing the first slice (i.e. the one starting at offset 0) is as fast as it can be, but computing the second slice involves the recomputation of the first slice and, more generally, computing the *nth* slice involves the recomputation of all preceding slices, the results of which are then just thrown away. This, of course, is a waste of resources and puts an unnecessary burden on the node.

This is not as bad as it looks. Most uses of `rpc/2-3` will compute all solutions at once and thus make only one network roundtrip.

```
?- time(rpc($_URI, (sleep(1), X=foo ; X=bar))).
% 2,011 inferences, 0.001 CPU in 1.007 seconds
X = foo ;
% 5 inferences, 0.000 CPU in 0.000 seconds
X = bar.
?-
```

It is only when the `limit` option must be employed, so that more than one network roundtrip has to be made, that the problem surfaces.

```
?- _Goal = (sleep(1), X=foo ; X=bar),
   time(compute_answer(_Goal, X, 0, 2, Answer)).
% 29 inferences, 0.000 CPU in 1.002 seconds
Answer = success([foo, bar], false).
?-
```

Still, to achieve a less wasteful and more efficient stateless querying even when more than one network roundtrip must be made, recomputation of the kind described in the previous section should be avoided. In this section we lay out an approach where the node controller (subject to a setting) may *cache* the state of the toplevel process that produced the *nth* slice of solutions to a query, so that the work spent on producing it will not have to be repeated. This can still be done without requiring that the node controller remembers *which* client made the request for the previous slices of solutions.

The method can be seen as a kind of *pooling* of toplevel processes, but while pooling usually involves a pool of merely initialized but idle processes which stand ready to be given work, this method involves a pool where each member has already done some real work. In other words, the idea here is not to cache *already computed* solutions but rather to cache the *potential* for new solutions in the form of processes that are idle, but have "more to give" if put to work.[5]

A consequence of this approach is that it allows the computation of the full set of solutions to a query to be distributed over more than one toplevel process. We can avoid spawning a new process for each incoming request, but instead, when available, select a member from a pool of suspended processes which, since it has already performed some of the work, needs to do as little as possible in order to compute the requested solutions. Using this approach, it is likely (although not guaranteed) that the work that generated the *nth* slice of solutions does not have to be repeated if a request for the next slice is made.

One way to realize this is to make the node controller responsible for the maintenance of a cache consisting of entries pointing to members of the pool of suspended processes. Such a cache has a very straightforward implementation in Prolog thanks to its dynamic database. The signature of a cache entry can be given as follows:

```
cache(+Gid, +N, -Pid) is nondet.
```

Here, `Gid` is an identifier representing a goal `G` and a template `T`. `N` is an integer > `0`, and `Pid` is the pid of an already spawned process which, after having computed `N` solutions to `G` and returned them to the client, is now suspended but can be activated again at any point. A cache is simply a dynamic predicate comprising an ordered sequence of `cache/3` clauses. The cache will be searched from the top, stopping when the first match is found. Updates will be added to the bottom.[6]

The cache forms a queue-like data structure and can be seen as a kind of priority queue. When a request comes in which specifies a goal, a template, and an offset > 1,

[5] Credits for this idea goes to Jan Wielemaker. The implementation is our's.

[6] Note that the implementation of the cache as a Prolog predicate is not mandated. A node would be free to implement it in a way that suits the host platform best.

the cache is scanned from the beginning of the queue, the first matching entry is dequeued, and the corresponding process is employed. If no matching entry is found, a new process is spawned. Newly created as well as updated cache entries are added to the end of the queue.

The maximum size of the cache for a particular node can be specified by its owner by means of a setting. What is a reasonable size depends on the host platform of the node, and in particular on the cost of keeping suspended toplevel actors around.

Here is an implementation of two predicates for managing the cache:

```
:- dynamic cache/3.

cache_retract(Gid, N, Pid) :-
    once(retract(cache(Gid, N, Pid))).

cache_update(Gid, N, Pid) :-
    assertz(cache(Gid, N, Pid)),
    setting(cache_size, Size),
    predicate_property(cache(_,_,_),
                       number_of_clauses(N)),
    N > Size -> cache_retract(_,_,_) ; true.
```

To ensure efficient cache lookup, the goal identifier `Gid` is a hash value computed from a grounded copy of the goal. In SWI-Prolog, `goal_id/2` may be implemented as follows:

```
goal_id(GoalTemplate, Gid) :-
    copy_term(GoalTemplate, Gid0),
    numbervars(Gid0, 0, _),
    term_hash(Gid0, Gid).
```

Equipped with the above utility predicates, `compute_answer/5` can be implemented like so:

```
compute_answer(Goal, Template, Offset, Limit, Answer) :-
    goal_id(Goal-Template, Gid),
    (   cache_retract(Gid, Offset, Pid)
    ->  thread_self(Self),
        toplevel_next(Pid, [
            limit(Limit),
            target(Self)
        ])
    ;   toplevel_spawn(Pid, [session(false)]),
        toplevel_call(Pid, Goal, [
            template(Template),
            offset(Offset),
            limit(Limit)
        ])
```

```
    ),
    setting(timeout, Timeout),
    receive({
        success(Pid, Slice, true) ->
            Index is Offset + Limit,
            cache_update(Gid, Index, Pid),
            Answer = success(Slice, true);
        success(Pid, Slice, false) ->
            Answer = success(Slice, false);
        failure(Pid) ->
            Answer = failure;
        error(Pid, Error) ->
            Answer = error(Error)
    },[
        timeout(Timeout),
        on_timeout((Answer = error(timeout),
                    toplevel_exit(Pid, kill)))
    ]).
```

Given a goal and a template, a goal identifier `Gid` is computed. Since more than one client may request the same slice of solutions, the Gid is not unique. Based on the gid and the value of the `offset` parameter, an attempt to look up a cache entry pointing to a suitable toplevel process will be made. If this succeeds, `toplevel_next/2` will be called, which will compute an answer holding a slice of solutions no longer than the value of the `limit` parameter specifies. If it fails, a new toplevel will be spawned using `toplevel_spawn/3`, and `toplevel_call/3` will be called, which will compute the answer instead.

The answer term resulting from this is sent to the thread in which the request handler is running and can be caught by `receive/2`. Note that if the reception of the term takes too long, it will result in a timeout error.

```
?- _Goal = (sleep(1), X=foo ; X=bar),
   time(compute_answer(_Goal, X, 0, 1, Answer)).
% 30 inferences, 0.000 CPU in 1.005 seconds
Answer = success([foo], true).
?-
```

...

```
?- _Goal = (sleep(1), X=foo ; X=bar),
   time(compute_answer(_Goal, X, 1, 1, Answer)).
% 30 inferences, 0.000 CPU in 0.005 seconds
Answer = success([bar], false).
?-
```

How can we extend the implementation of the ISOBASE node so that it can serve also as an ISOTOPE node? As evident from the diagram in Figure 4.3, it needs support for the `load_text` parameter. Its value must be sent along when calling `toplevel_spawn/2`, which will inject the code in the private database of the toplevel. Moreover, the goal identifier must be based on *both* the goal, the template and this value. Code for handling all of this would be easy to add.

A.1.4 Implementing the Erlang-style concurrency predicates

This section implement specifications for how we believe predicates such as `spawn/2-3`, `exit/1-2`, `!/2` and `receive/1-2` might work. To keep things as succinct as possible we do not add code checking the instantiation of arguments. (However, some such tests are present in the proof-of-concept mini implementation.)

Today widely available Prolog systems can be differentiated whether they are multi-threaded or not. In a multi-threaded Prolog system we can create multiple threads that run concurrently over the same knowledge base. From Table 2 in *Fifty Years of Prolog and Beyond* we learn that out of the Prolog systems listed above, five implement multi-threading support. According to this table, these are Ciao, ECLiPSe, SWI-Prolog, tuProlog and XSB. However, we have found that Trealla Prolog should also be added to the list, and thus we have six systems with multi-threading support.

There is a draft standard for multi-threading support in Prolog, specified in a document that begins like so:

> ISO/IEC DTR 13211–5:2007 Prolog multi-threading support [...] is an optional part of the International Standard for Prolog, ISO/IEC 13211. [...] Multi-thread predicates are based on the semantics of POSIX threads. They have been implemented in some Prolog systems. As such, they are deemed a worthy extension to the ISO/IEC 13211 Prolog standard.[7]

Except for Ciao Prolog, which takes a different approach to multi-threading, the six systems listed above all implement the draft standard.

In order to support the Erlang-style concurrency predicates offered by the ACTOR profile of Web Prolog the five predicates on the left can be implemented by means of the seven predicates from the draft standard on the right:

```
spawn/3              thread_create/3
self/1               thread_self/1
!/2, send/2          thread_send_message/2
receive/1-2          thread_get_message/3
exit/2               thread_signal/2
                     thread_detach/1
                     thread_property/2
```

The drafts standard specifies more than a dozen more predicates, such as predicates for creating message queues and managing mutexes. We do not need those.

Here is a first sketch of an implementation of `spawn/2-3`:

[7] https://logtalk.org/plstd/threads.pdf

```
:- op(800,  xfx, !).
:- op(1000, xfy, when).

:- dynamic link/2.

spawn(Goal) :-
    spawn(Goal, _Pid).

spawn(Goal, Pid) :-
    spawn(Goal, Pid, []).

spawn(Goal, Pid, Options) :-
    thread_self(Self),
    make_pid(Pid),
    thread_create(start(Self, Pid, Goal, Options), Pid, [
        alias(Pid),
        at_exit(stop(Pid, Self))
    ]),
    thread_get_message(initialized(Pid)).


make_pid(Pid) :-
    random_between(10000000, 99999990, Num),
    atom_number(Pid, Num).


:- thread_local parent/1.

start(Parent, Pid, Goal, Options) :-
    assertz(parent(Parent)),
    option(link(Link), Options, true),
    (   Link == true
    ->  assertz(link(Parent, Pid))
    ;   true
    ),
    option(monitor(Monitor), Options, false),
    (   Monitor == true
    ->  assertz(monitor(Parent, Pid))
    ;   true
    ),
    thread_send_message(Parent, initialized(Pid)),
    call(Goal).


stop(Pid, Parent) :-
```

```
    thread_detach(Pid),
    retractall(link(Parent, Pid)),
    retractall(registered(_Name, Pid)),
    forall(retract(link(Pid, ChildPid)),
           exit(ChildPid, linked)),
    down_reason(Pid, Reason),
    forall(retract(monitor(Other, Pid)),
           Other ! down(Pid, Reason)).

down_reason(Pid, Reason) :-
    retract(exit_reason(Pid, Reason)),
    !.
down_reason(Pid, Reason) :-
    thread_property(Pid, status(Reason)).
```

A thread implements an actor. The thread comes with its own message queue, which will serve as the actor's mailbox. The thread identifier works like a pid.

A number of thread-related predicates are called that finds the identity of the soon-to-become parent, creates a thread that, just before terminating, calls `down/3`, which takes care of what must be done in the last moment before the actor terminates – the termination of any children that it may have spawned during its life cycle (in case `link` is set to `true`), and the sending of a `down` message to the parent (if `monitor` is set to `true`).

The above implementation of `spawn/2-3` calls two predicates – `exit/2` and `!/2` – that must be implemented. In addition, `exit/1` must be implemented, and this can be done as followws:

```
:- dynamic exit_reason/2.

exit(Reason) :-
    self(Self),
    asserta(exit_reason(Self, Reason)),
    abort.
```

For the implementation of `exit/2`, ISO/IEC DTR 13211–5:2007 specifies a predicate `thread_signal/2` to make a thread execute some goal as an interrupt. Signaling may be used to cancel no-longer-needed threads. This means that `exit/2` may be implemented like so:

```
exit(Pid, Reason) :-
    catch(thread_signal(Pid,
          exit(Reason)),
          error(existence_error(_,_), _),
          true).
```

Note that `thread_signal/2` throws an error if the thread ID in the first argument points to a thread that does not exist. Since `exit/2` must succeed also in this case, we have wrapped the call to `thread_signal/2` in a call to `catch/3`.

For the implementation of `!/2`, ISO/IEC DTR 13211–5:2007 offers a predicate `thread_send_message/2` which is somewhat similar to Erlang's send primitive. It allows any term to be sent to any thread. Just like in Erlang, the term is copied to the receiving process and variable bindings are thus lost. However, `thread_send_message/2` throws an error if the thread ID in the first argument points to a thread that does not exist. Again, since `!/2`, just like in Erlang, should succeed also in this case, we wrap the call in `catch/3` like so:

```
Pid ! Message :-
    send(Pid, Message).

send(Name, Message) :-
    registered(Name, Pid),
    !,
    send(Pid, Message).
send(Pid, Message) :-
    catch(thread_send_message(Pid, Message),
          error(existence_error(_,_), _),
          true).
```

In effect, this makes any attempt to send a message to a non-existing actor a no-op.

The predicates `output/1-2`, `input/2-3` and `respond/2` are implemented on top of the !/2 primitive. Their purpose is to simulate I/O.

Here is the suggested implementation of `output/1-2`:

```
output(Term) :-
    output(Term, []).

output(Term, Options) :-
    self(Self),
    parent(Parent),
    option(target(Target), Options, Parent),
    Target ! output(Self, Term).
```

The implementation of `input/2-3` is slightly more complicated:

```
input(Prompt, Input) :-
    input(Prompt, Input, []).

input(Prompt, Input, Options) :-
    self(Self),
    parent(Parent),
    option(target(Target), Options, Parent),
    Target ! prompt(Self, Prompt),
    receive({
       '$input'(Target, Input) ->
           true
    }).
```

The predicate `respond/2` is used to respond to a prompt:

```
respond(Pid, Term) :-
    self(Self),
    Pid ! '$input'(Self, Term).
```

The implementation of the receive operation is somewhat more involved. Relying on `thread_get_message/3`, what might be regarded as a reference implementation of `receive/1-2` looks like this:

```
:- thread_local deferred/1.

receive(Clauses) :-
    receive(Clauses, []).

receive(Clauses, Options) :-
    thread_self(Mailbox),
    (   clause(deferred(Msg), true, Ref),
        select_body(Clauses, Msg, Body)
    ->  erase(Ref),
        call(Body)
    ;   receive(Mailbox, Clauses, Options)
    ).

receive(Mailbox, Clauses, Options) :-
    (   thread_get_message(Mailbox, Msg, Options)
    ->  (   select_body(Clauses, Msg, Body)
        ->  call(Body)
        ;   assertz(deferred(Msg)),
            receive(Mailbox, Clauses, Options)
        )
    ;   option(on_timeout(Body), Options, true),
        call(Body)
    ).

select_body(_M:{Clauses}, Message, Body) :-
    select_body_aux(Clauses, Message, Body).

select_body_aux((Clause ; Clauses), Message, Body) :-
    (   select_body_aux(Clause,  Message, Body)
    ;   select_body_aux(Clauses, Message, Body)
    ).
select_body_aux((Head -> Body), Message, Body) :-
    (   subsumes_term(if(Pattern, Guard), Head)
    ->  if(Pattern, Guard) = Head,
        subsumes_term(Pattern, Message),
```

```
        Pattern = Message,
        catch(once(Guard), _, fail)
    ;   subsumes_term(Head, Message),
        Head = Message
    ).
```

A.1.5 Implementing the first-class Prolog toplevel

In addition to the Erlang-style actors, the toplevel behavior, controlled by predicates such as `toplevel_spawn/1-2` and friends, must also be implemented. We refer the reader back to Chapter 3 for how this should work and for some hints for how it can be implemented. In our experience, once we have a complete implementation of all the Erlang-style primitives for concurrency and distribution, the implementation of the toplevel behavior and the built-in predicates for controlling it is fairly straightforward.

We begin with an implementation of `toplevel_spawn/1-2`:

```
toplevel_spawn(Pid) :-
    toplevel_spawn(Pid, []).

toplevel_spawn(Pid, Options) :-
    self(Self),
    option(session(Session), Options, false),
    option(target(Target), Options, Self),
    spawn(state_1(Pid, Target, Exit), Pid, Options).
```

Note that options passed to `toplevel_spawn/2` will be passed on to `spawn/3` as well.

The most important part of the implementation of the PTCP protocol are the three states s1, s2 and s3 , depicted in the diagram in Figure A.1:

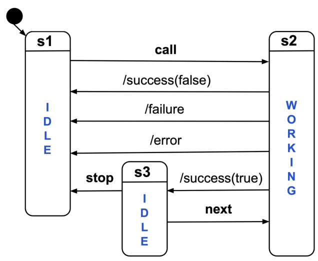

Fig. A.1 The three inner states of the PTCP protocol.

```
state_1(Pid, Target0, Session) :-
```

```
    receive({
        '$call'(Goal, Options) ->
            option(template(Template), Options, Goal),
            option(offset(Offset), Options, 0),
            option(limit(Limit0), Options, 10 000 000 000),
            option(target(Target1), Options, Target0),
            Limit = count(Limit0),
            state_2(Goal, Template, Offset, Limit, Pid, Answer),
            Target = target(Target1),
            arg(1, Target, Out),
            Out ! Answer,
            (   arg(3, Answer, true)
            ->  state_3(Limit, Target)
            ;   true
            )
        }),
    (   Session == false
    ->  true
    ;   state_1(Pid, Target0, Session)
    ).
```

In `state_2` the real work is being done. The predicate `answer/5`, defined earlier in this chapter in the context of the stateless web API, is reused. However, answer terms must be extended with the pids of the actor processes that produced them.

```
state_2(Goal, Template, Offset, Limit, Pid, Answer) :-
    answer(Goal, Template, Offset, Limit, Answer0),
    add_pid(Answer0, Pid, Answer).
```

To handle this, `add_pid/3` is defined like so:

```
add_pid(success(Slice, More), Pid, success(Pid, Slice, More)).
add_pid(failure, Pid, failure(Pid)).
add_pid(error(Term), Pid, error(Pid, Term)).
```

One feature of `answer/5` that was not demonstrated before, is that the argument specifying the limit can be passed a unary term `count` with an integer in its argument. This works like a mutable local variable that can be assigned values using `nb_setarg/3` and read by means of `arg/3`.

```
?- Limit = count(2),
   answer(between(1,12,I), I, 0, Limit, Answer),
   nb_setarg(1, Limit, 5).
Limit = count(5),
Answer = success([1, 2], true) ;
Limit = count(5),
Answer = success([3, 4, 5, 6, 7], true) ;
```

```
Limit = count(5),
Answer = success([8, 9, 10, 11, 12], false).
?-
```

In the definition of the predicate `state_1/3` we saw that if a `success` answer term indicates (with `true` in its third argument) that there may be more solutions to the current goal, we enter `state_3`. For other answer terms a recursive call of `state_1/3` is made.

```
state_3(Limit, Target) :-
    receive({
        '$next'(Options2) ->
            (   option(limit(NewLimit), Options2)
            ->  nb_setarg(1, Limit, NewLimit)
            ;   true
            ),
            (   option(target(NewTarget), Options2)
            ->  nb_setarg(1, Target, NewTarget)
            ;   true
            ),
            fail ;
        '$stop' -> true
    }).
```

Here it is the reception of the `'$next'` message and the subsequent call to `fail/0` that triggers the backtracking to `answer/5` in state s2. If the `'$stop'` message is received instead, `state_3/2` terminates, and then `state_1/3` terminates too (unless the option `session(true)` was passed to `toplevel_spawn/1-2`).

As can be seen in the diagram depicting the PTCP, we have so far only implemented the three inner states.

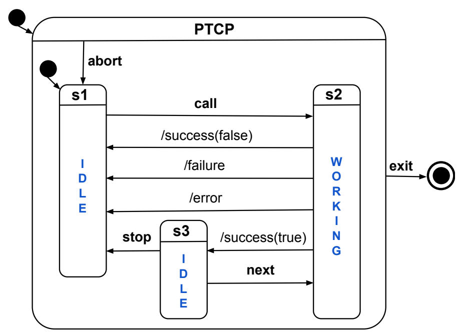

Fig. A.2 The complete PTCP protocol.

We need to enable a client to abort the execution of a goal:

```
ptcp(Pid, Target, Session) :-
    catch(state_1(Pid, Target, Session),
          '$abort_goal',
          ptcp(Pid, Target, Session)).
```

To make it work, the last line in the above implementation of `toplevel_spawn/2` must be changed into this:

```
    spawn(ptcp(Pid, Target, Session), Pid, Options).
```

Here is to tell the toplevel actor to abort the execution of any goal that it currently runs:

```
toplevel_abort(Pid) :-
    catch(thread_signal(Pid, throw('$abort_goal')),
          error(existence_error(_,_), _),
          true).
```

The action of aborting a particular execution of a goal passed to `toplevel_call/2-3` must not be confused with the action of exiting the toplevel process. The latter can be performed by using `toplevel_exit/2` (or just `exit/2` which as can be seen here means the same):

```
toplevel_exit(Pid, Reason) :-
    exit(Pid, Reason).
```

As suggested already in Chapter 2, programmers should not be burdened with having to remember the details of protocols and forms of built-in messages such as `'$call'`, `'$next'` and `'$stop'`. Instead, such details should be hidden behind interface predicates dealing with sending them, implementing `toplevel_call/2-3` simply as

```
toplevel_call(Pid, Goal) :-
    toplevel_call(Pid, Goal, []).

toplevel_call(Pid, Goal, Options) :-
    Pid ! '$call'(Goal, Options).
```

and `toplevel_next/1-2` like so

```
toplevel_next(Pid) :-
    toplevel_next(Pid, []).

toplevel_next(Pid, Options) :-
    Pid ! '$next'(Options).
```

and, finally, `toplevel_stop/1` like so:

```
toplevel_stop(Pid) :-
    Pid ! '$stop'.
```

A.1.6 What is missing from the sketches?

The predicates implemented so far are sufficient for running many of the example programs given in Chapter 2 and Chapter 3 of this book. Of course, this is just a start, and to be able to run *all* programs, and in particular the ones in Chapter 4, more is needed. Notably, the current implementation sketch does not support

- network-transparent concurrency and distribution,
- the implementation of an actors's private database, and
- security.

As for network transparency, the scenarios in Chapter 4 show in great detail how the stateful distribution layer might work. Recall that to spawn an actor on a remote node, the `node` option must be passed to `spawn/3` with a URI pointing to the node:

```
?- spawn(foo, Pid, [
       node('http://n7.org')
   ]).
Pid = 34925412@'http://n7.org'.
?-
```

Note that once this works for `spawn/3`, it will work for `toplevel_spawn/2` too.
Exiting remote processes must also be implemented so that it can be handled in the following way:

```
?- exit(34925412@'http://n7.org', normal).
true.
?-
```

Our implemention of the send operator will only work for the simplest of cases of local messaging, but a complete implementation of an ACTOR node must also allow sending to remote processes, like so:

```
?- 34925412@'http://n7.org' ! bar.
true.
?-
```

Once this works for `!/2`, it will also make `toplevel_call/2-3`, `toplevel_next/1-2` and related predicates work.
Note that the stateful distribution layer depends on WebSockets and that, as far as we know, at this point in time SWI-Prolog is the only Prolog system that offers a WebSocket library.
Source code injection such as in the following example must also be supported by an ACTOR node:

```
?- spawn(baz, Pid, [
      load_text('p(a). p(b).')
   ]).
```

```
Pid = 71123976@'http://n1.org'.
?-
```

Injected source code must end up in the spawned actor's private Prolog database and thus we need a viable approach to the implementation of this database and the isolation it requires. Isolation can be based on `thread_local/1` or the use of temporary modules. (Temporary modules are used by `library(pengines)`.)

If source code injection works for `spawn/3`, it will work for `toplevel_spawn/2` and `rpc/3` as well.

On the subject of security, a very important requirement relates to *sandboxing*. The approach taken by `library(sandbox)` in SWISH is not satisfactory.