The role of the raven in the Prolog Trinity ecosystem

Scryer Prolog agents on the Web

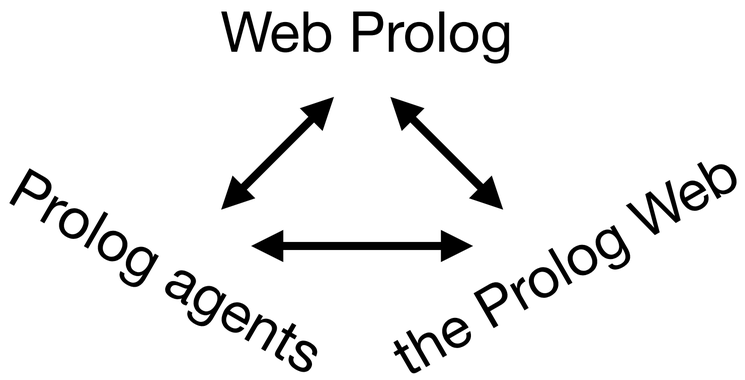
Deep into that darkness peering, long I stood there, wondering, fearing,
doubting, dreaming dreams no mortal ever dared to dream before.

— Edgar Allan Poe, The Raven

Torbjörn Lager

Department of Philosophy, Linguistics and Theory of Science
University of Gothenburg
Sweden

Mail: torbjorn.lager@gu.se


Web Prolog
Prolog agents
the Prolog Web

Outline

Main talking points

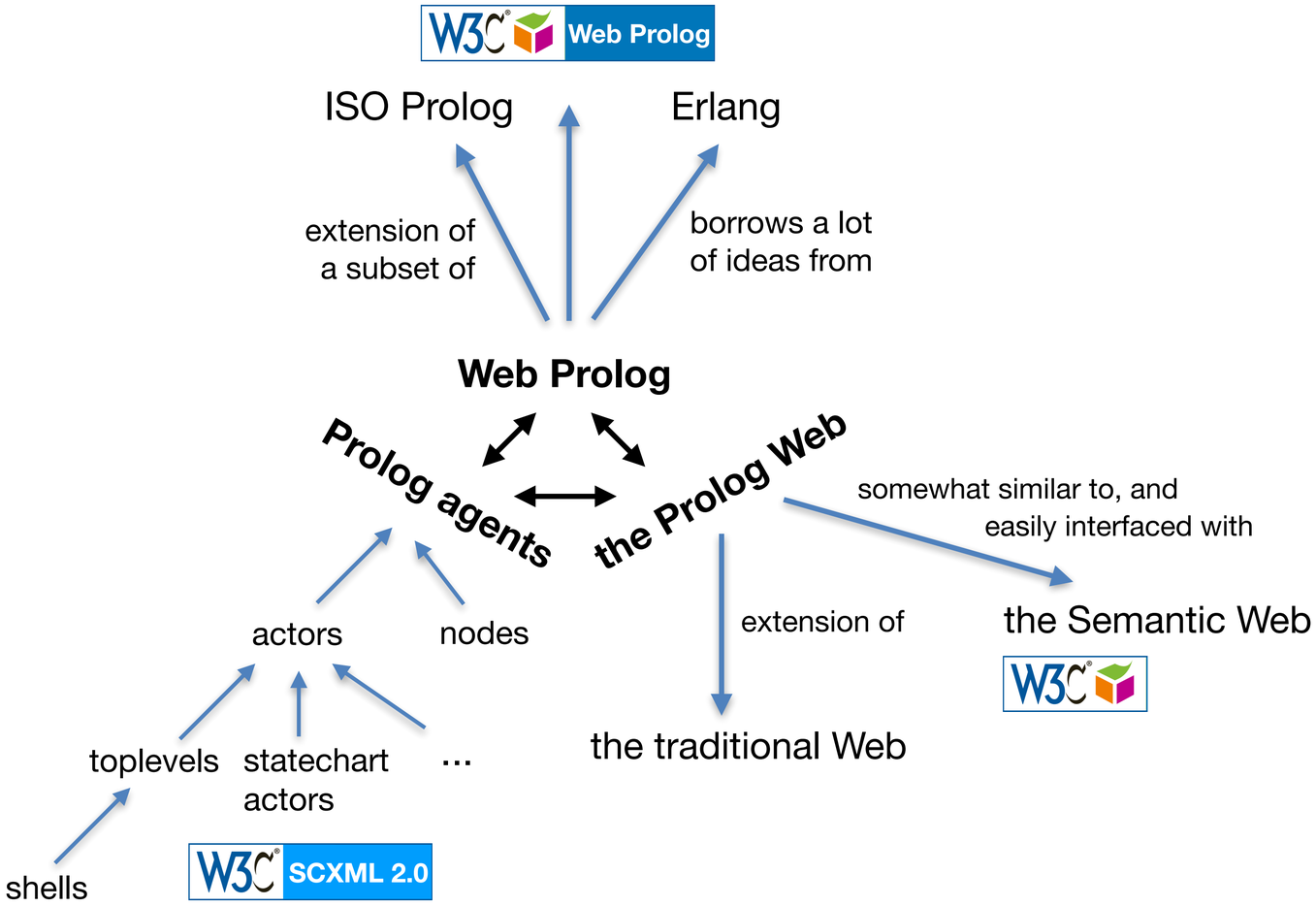
1. The Prolog Trinity ecosystem
2. Erlang-style programming of Prolog agents in Web Prolog
3. Building and deploying applications on the Prolog Web
4. Standardizing Web Prolog — and a path to ISO Prolog 2.0

Add-ons (if time permits)

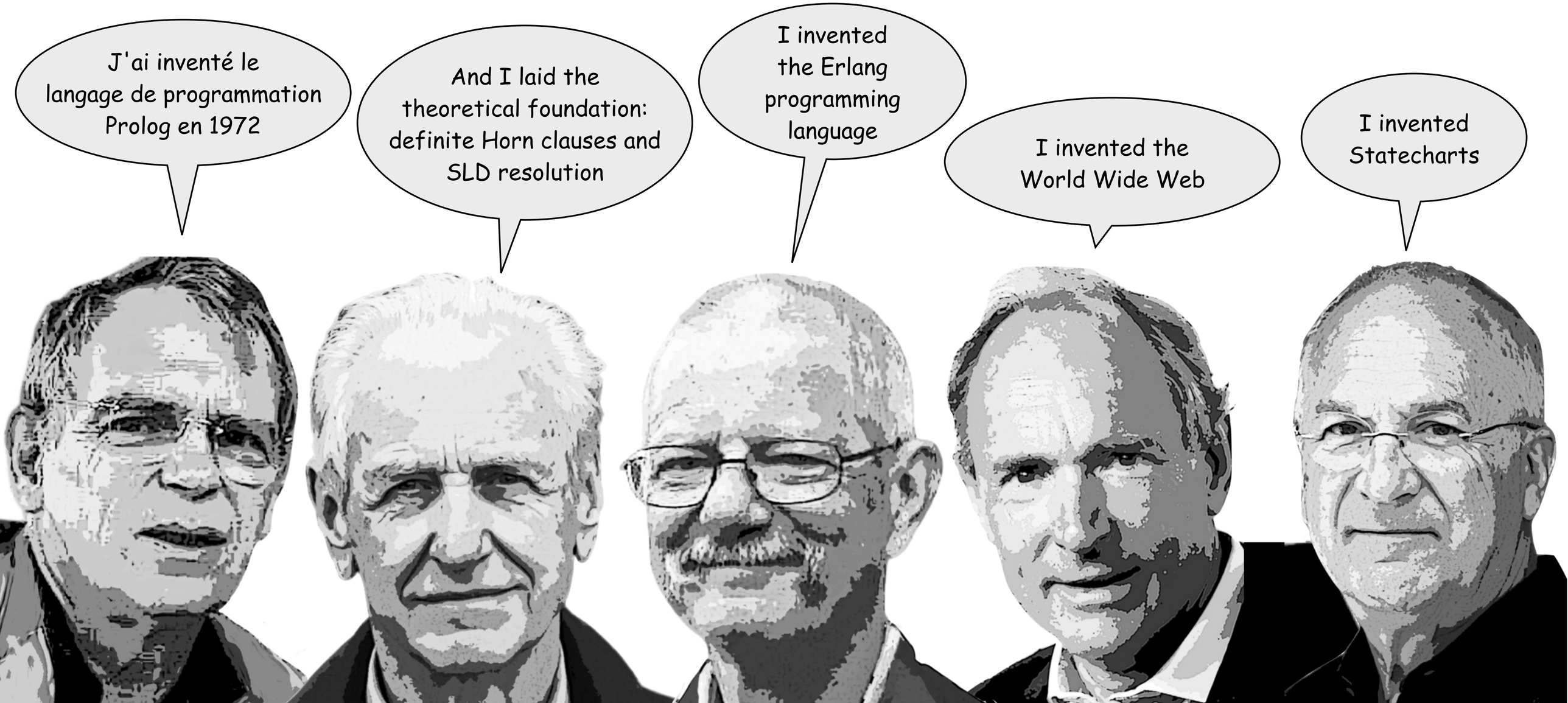
A. Implementing the stateless HTTP API and rpc/2-3
B. Avoiding spurious recomputation in the HTTP API
C. Implementing actors on top the Threads (draft) standard
D. Implementing first-class toplevels on top of actors

Other resources (at http://torbjornlager.github.io)

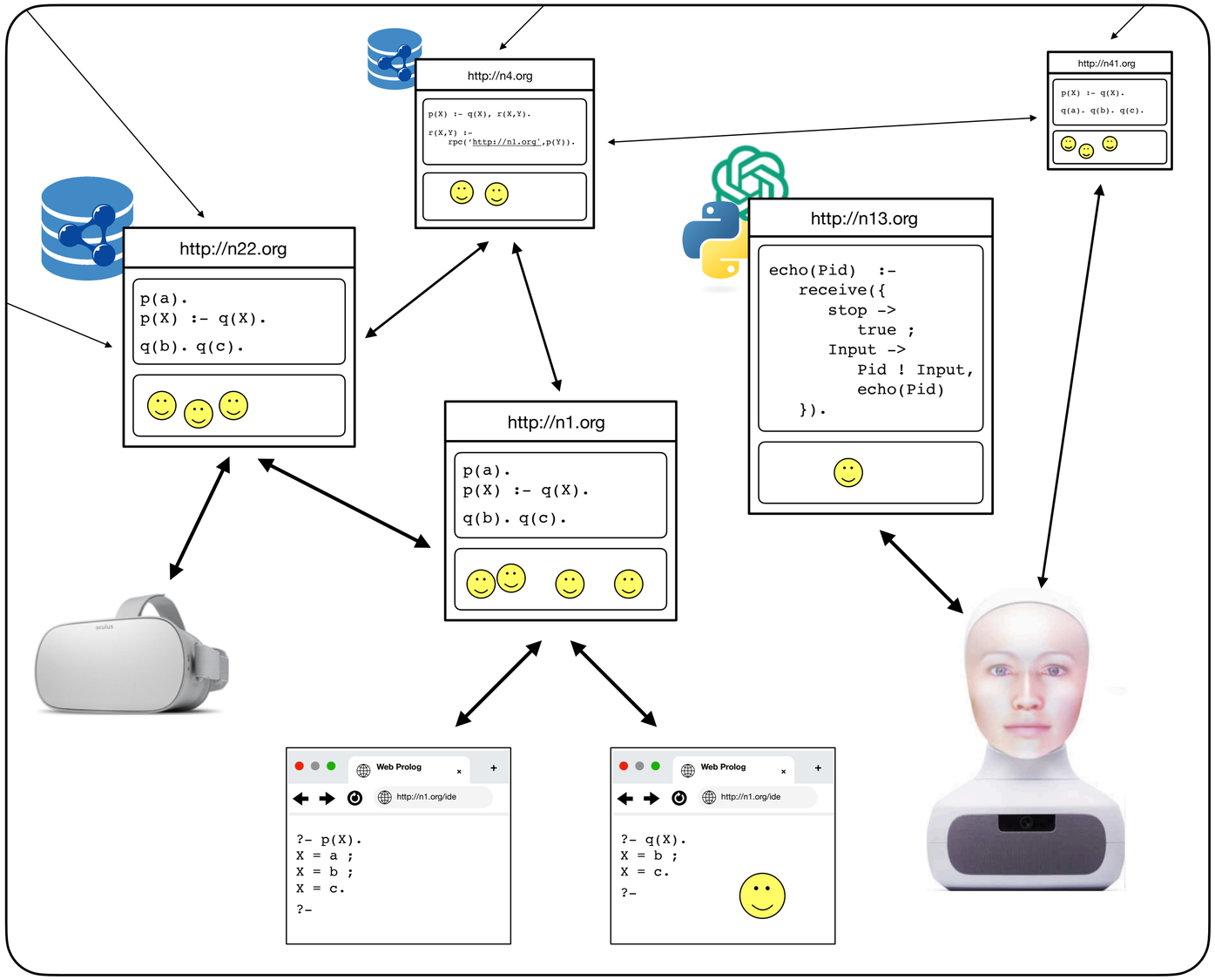
1. A proof-of-concept implementation of Web Prolog written for clarity, and
2. an appendix from my book manuscript detailing how it works

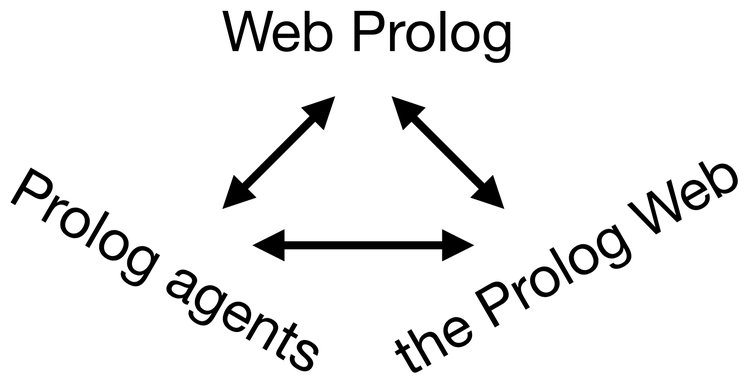

Web Prolog
ISO Prolog
Erlang
extension of
a subset of
borrows a lot
of ideas from
Web Prolog
Prolog agents
the Prolog Web
somewhat similar to, and
easily interfaced with
actors
nodes
extension of
the Semantic Web
toplevels
statechart
actors
...
the traditional Web
shells
SCXML 2.0

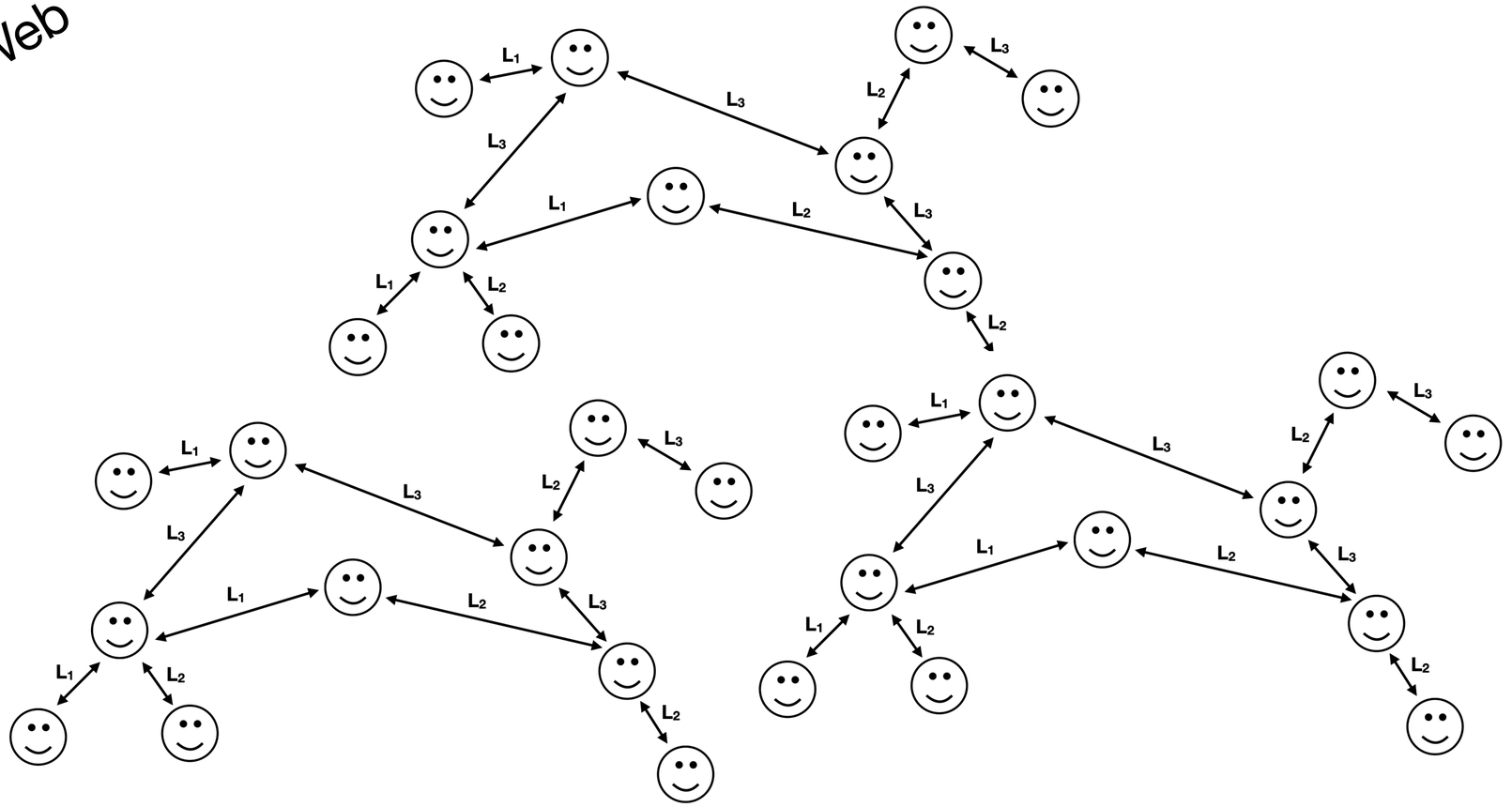
Walking in the footsteps of inventors



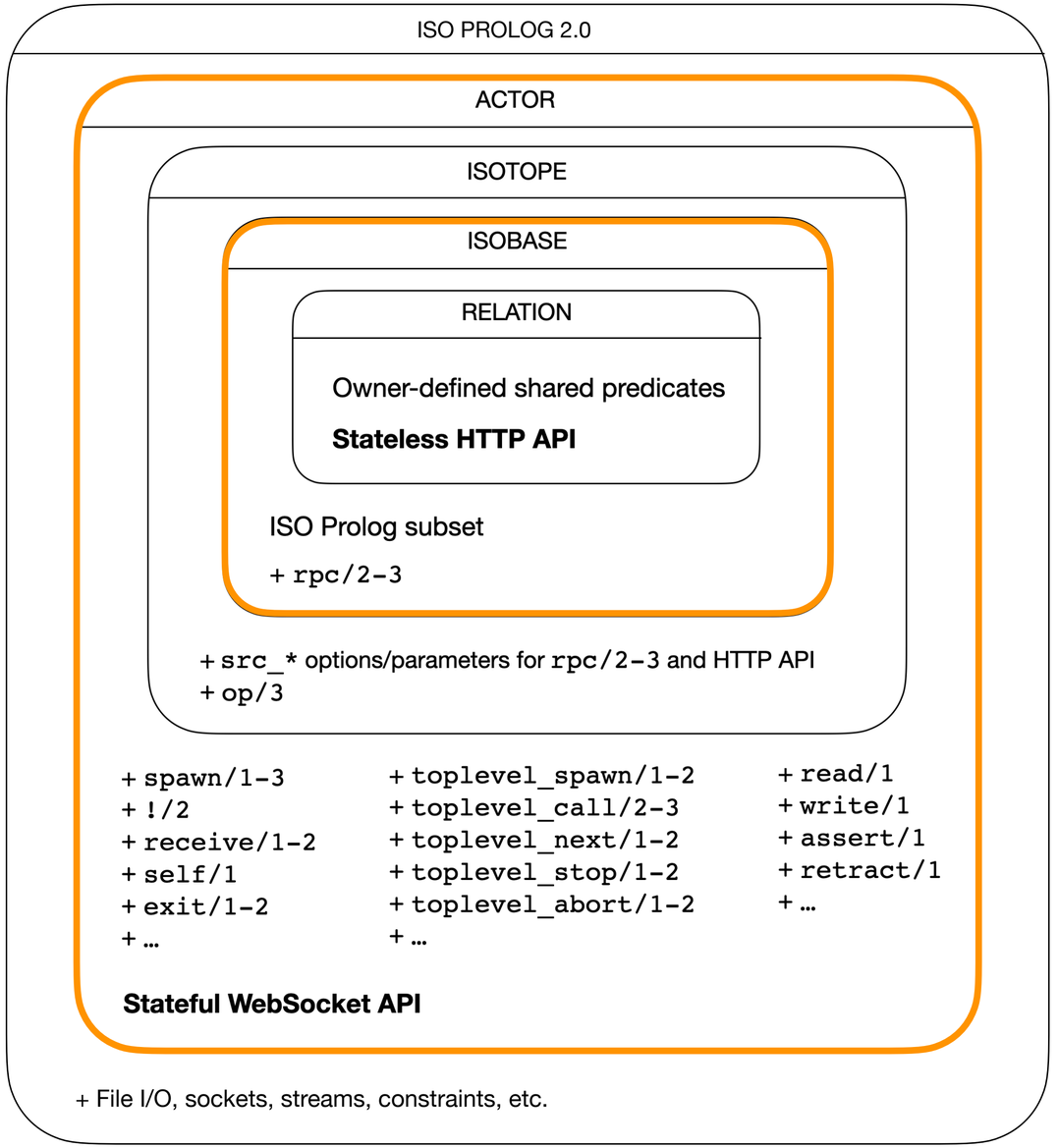
A Web of Prolog agents




Web Prolog
Prolog agents
the Prolog Web


L1
L2
L3

Node profiles: The ISOBASE profile

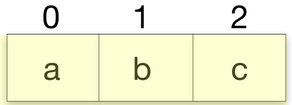


The stateless HTTP API

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

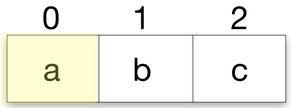
```
GET http://n1.org/call?goal=p(X)
```

```
{"type": "success",
 "data": [{"X":"a"},{"X":"b"},{"X":"c"}],
 "more": false}
```



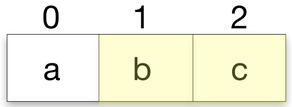

```
GET http://n1.org/call?goal=p(X)&limit=1
```

```
{"type": "success",
 "data": [{"X":"a"}],
 "more": true}
```



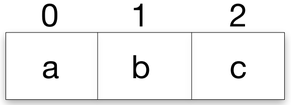

```
GET http://n1.org/call?goal=p(X)&offset=1
```

```
{"type": "success",
 "data": [{"X":"b"},{"X":"c"}],
 "more": false}
```



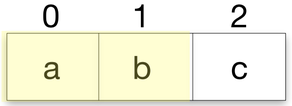

```
GET http://n1.org/call?goal=p(X)&offset=15&limit=10
```

```
{"type": "failure"}
```




```
GET http://n1.org/call?goal=p(X)&limit=2&format=prolog
```

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



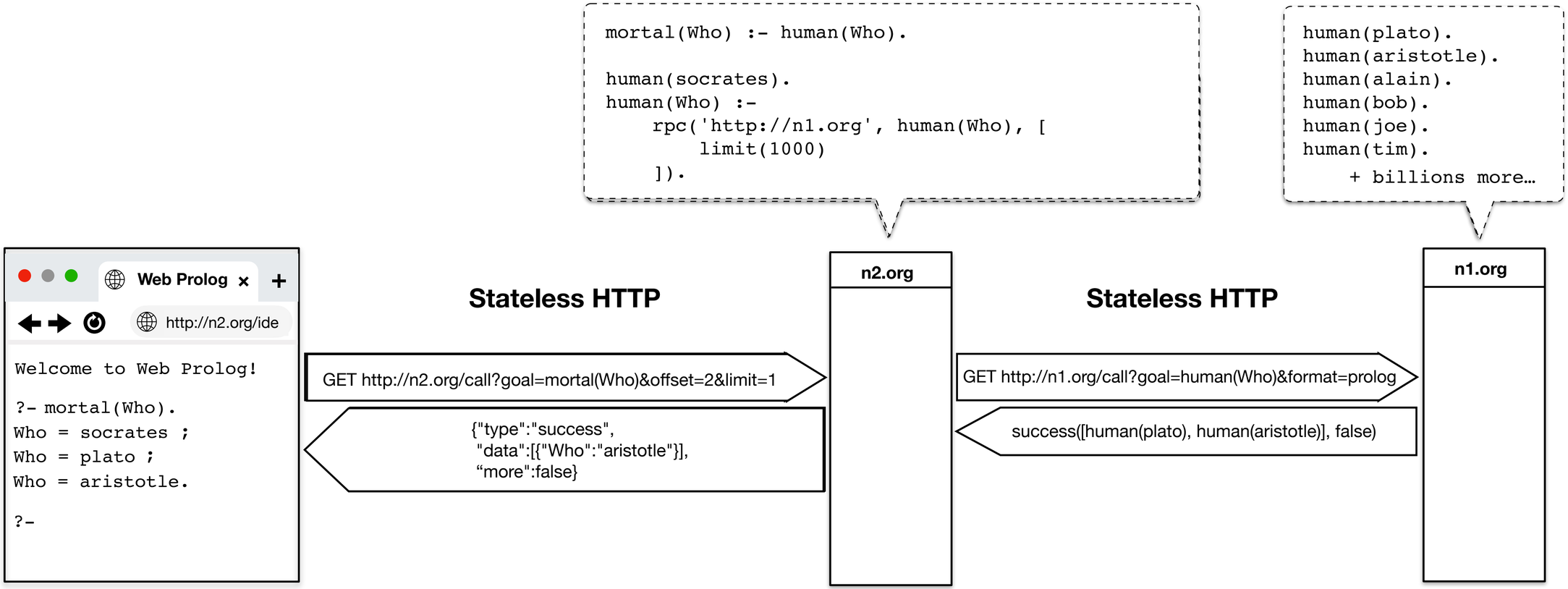
Non-deterministic RPC

- The predicate `rpc/2-3` allows a process running at a node A to call a goal in the Prolog context of another node B, taking advantage of the programs and data being offered by B, just as if they were local to A.

```
rpc(+URI,:Goal) is nondet.
rpc(+URI,:Goal,+Options) is nondet.
```

- Very simple and easy to use — no concurrency involved.
- Purity preserving.
- Can be implemented on top of the stateless HTTP API.

Browser talking to a node talking to a node



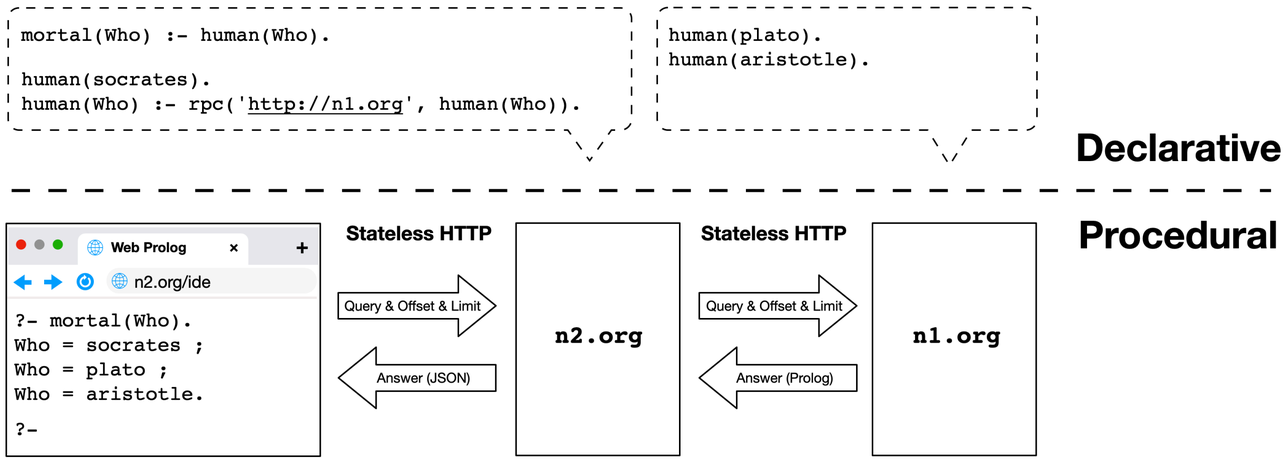
The pure Prolog Web

Two definitions:

Pure Web Prolog = pure Prolog + `rpc/2-3`

The Pure Prolog Web = the partitions of the Prolog Web written in pure Web Prolog

Two levels of abstraction:



How can that be?:

Because `rpc/2-3` retains the logical purity of the goal that it calls, so that if the goal is pure, then the entire call is pure. Therefore, `rpc('http://n1.org',human(Who))` is pure!


$(p \land q) \to r$
$\neg p \lor \neg q \lor \neg r \lor s$
$p \to q$
$\neg p \lor q$
$\neg p \lor \neg q \lor r$
$\neg p \lor q$
$(p \land q) \to r$
$\neg p \lor q$
$(p \land q) \to r$
$\neg p \lor \neg q \lor r$
$(p \land q) \to r$
$\neg p \lor \neg q \lor \neg r \lor s$
$p \to q$
$(p \land q \land r) \to s$
$\neg p \lor q$
$\neg p \lor \neg q \lor r$
$p \to q$
$(p \land q) \to r$
$(p \land q \land r) \to s$
$\neg p \lor \neg q \lor r$
$\neg p \lor q$
$p \to q$
$(p \land q \land r) \to s$
$\neg p \lor \neg q \lor \neg r \lor s$
$(p \land q) \to r$
$p \to q$
$\neg p \lor q$
$(p \land q) \to r$
$\neg p \lor \neg q \lor \neg r \lor s$
$(p \land q \land r) \to s$
$\neg p \lor q$
$p \to q$
$(p \land q \land r) \to s$
$\neg p \lor q$
$\neg p \lor \neg q \lor r$
$p \to q$
$(p \land q \land r) \to s$
$(p \land q) \to r$
$(p \land q) \to r$
$p \to q$
$\neg p \lor q$
$(p \land q) \to r$

The stateless API — pros and cons

Pros

- Supports retrieval of solutions one-at-a-time in interactions such as this:

```
?- mortal(Who).
Who = socrates ;
Who = plato ;
Who = aristotle.

?-
```

- Instead of having each client tie up part of a node's resources for as long as a session lasts, it can release them as soon as it wants.

- Statelessness makes *caching* of responses to queries by intermediates possible and often worthwhile.

Cons

- Once a client has submitted a goal, there is no way it can abort it.

```
?- repeat, fail.
Error: Timeout exceeded

?-
```

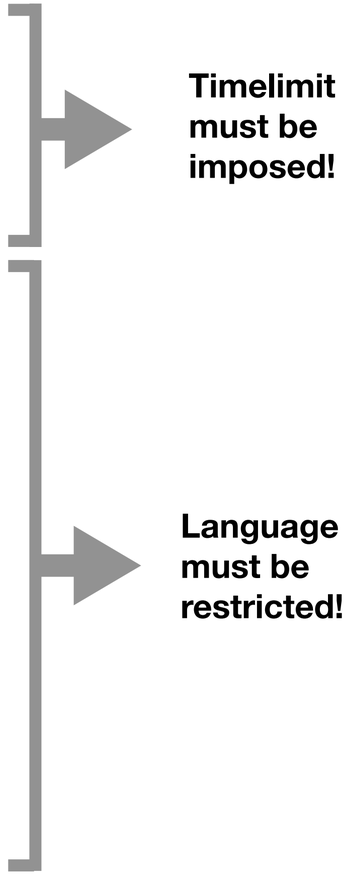
- Supporting I/O over a stateless API isn't really possible.

```
?- writeln(hello).
true.

?-
```

- Database updates using assert and retract are often not possible.

```
?- assert(p(a)).
true.

?- retract(p(X)).
false.

?-
```

- Actor programming cannot be supported.

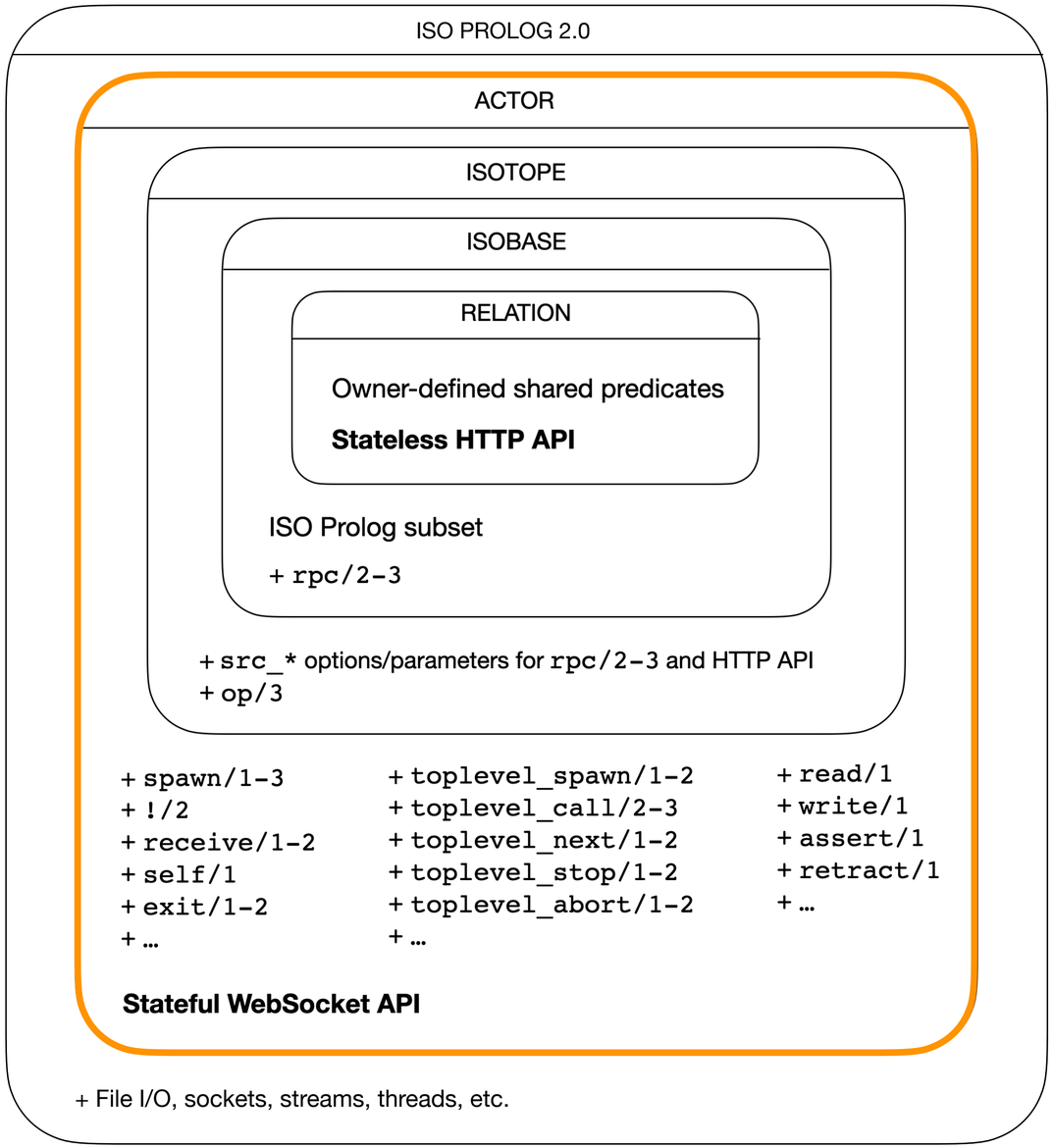


Scryer Prolog agents on the Web

Can Scryer Prolog, already at this point of its development, implement an ISOBASE node?

Yes! **But...**

Node profiles: the ACTOR profile



Theory and Practice of Logic Programming

Fifty Years of Prolog and Beyond *

PHILIPP KÖRNER, MICHAEL LEUSCHEL
Institut für Informatik, Universität Düsseldorf, Universitätsstr. 1, D-40225 Düsseldorf
(*e-mail:* {p.koerner,leuschel}@uni-duesseldorf.de)

JOÃO BARBOSA, VÍTOR SANTOS COSTA
Department of Computer Science, Faculty of Science of the University of Porto
(*e-mail:* {joao.barbosa,vscosta}@fc.up.pt)

VERÓNICA DAHL
Computing Sciences Department, Simon Fraser University
(*e-mail:* veronica_dahl@sfu.ca)

MANUEL V. HERMENEGILDO, JOSE F. MORALES
IMDEA Software Institute and Universidad Politécnica de Madrid (UPM)
(*e-mail:* {manuel.hermenegildo,josef.morales}@imdea.org)

JAN WIELEMAKER
Centrum voor Wiskunder en Informatica (CWI), Amsterdam
(*e-mail:* J.Wielemaker@cwi.nl)

DANIEL DIAZ
Centre de Recherche en Informatique, University Paris-1
(*e-mail:* daniel.diaz@univ-paris1.fr)

SALVADOR ABREU
NOVA-LINCS, University of Évora
(*e-mail:* spa@uevora.pt)

GIOVANNI CIATTO
Dept. of Computer Science and Engineering, Alma Mater Studiorum—Univerità di Bologna
(*e-mail:* giovanni.ciatto@unibo.it)

Essential and important features of ISO Prolog 2.0

Essential and important features of ISO Prolog

1. Horn clauses with variables in the terms and arbitrarily nested function symbols as the basic knowledge representation means for both programs (a.k.a. knowledge bases) and queries;
2. the ability to manipulate predicates and clauses as terms, so that meta-predicates can be written as ordinary predicates;
3. SLD-resolution (Kowalski, 1974) based on Robinson's principle (1965) and Kowalski's procedural semantics (Kowalski, 1974) as the basic execution mechanism;
4. unification of arbitrary terms which may contain logic variables at any position, both during SLD-resolution steps and as an explicit mechanism (e.g., via the built-in =/2);
5. the automatic depth-first exploration of the proof tree for each logic query;
6. some control mechanism aimed at letting programmers manage the aforementioned exploration;
7. negation as failure (Clark, 1978), and other logic aspects such as disjunction or implication;
8. the possibility to alter the execution context during resolution, via ad-hoc primitives;
9. an efficient way of indexing clauses in the knowledge base, for both the read-only and read-write use cases;
10. the possibility to express definite clause grammars (DCG) and parse strings using them;
11. constraint logic programming (Jaffar and Lassez, 1987) via ad-hoc predicates or specialized rules (Fruhwirth, 2009);
12. the possibility to define custom infix, prefix, or postfix operators, with arbitrary priority and associativity.

Essential features of Erlang

1. The ability to execute a large number of actor processes *concurrently*;
2. the ability to use *message-passing* for inter-process communication, avoiding mutable shared memory and locking issues;
3. the ability to support *network-transparent* concurrent programming where actors are allowed to create other actors on remote computers and enter into seamless communication with them.

What we would get:

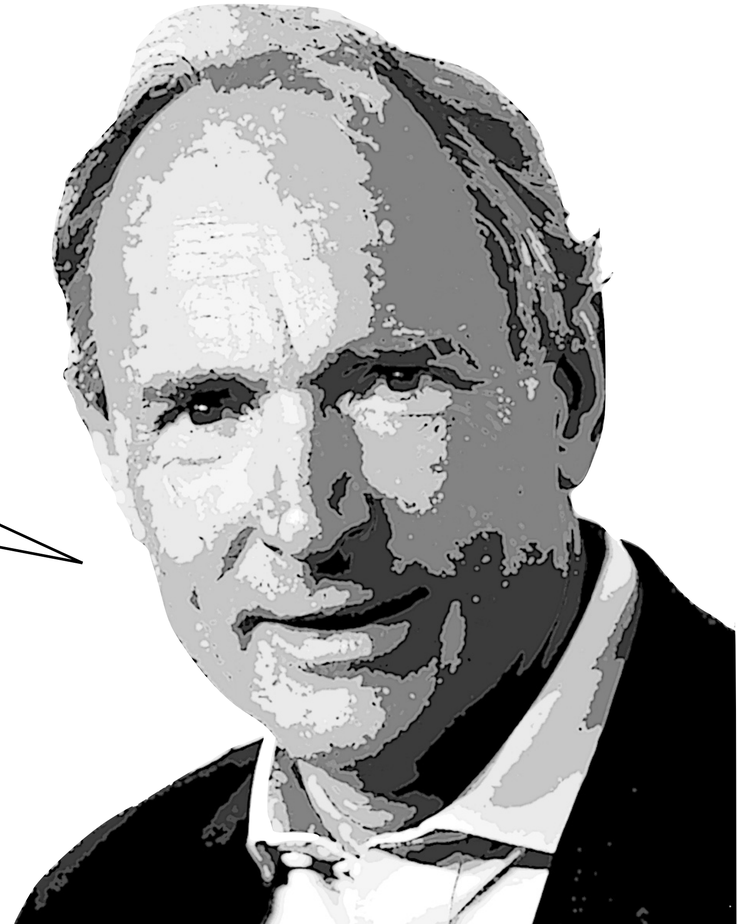
1. A multi-paradigm language
2. First-class toplevel actors
3. Real-time interaction over well-defined protocols
4. An effective software development methodology

   By decomposing complex problems into smaller, independent tasks, concurrency allows for modular system design. Each task can be developed, tested, and maintained separately, improving code

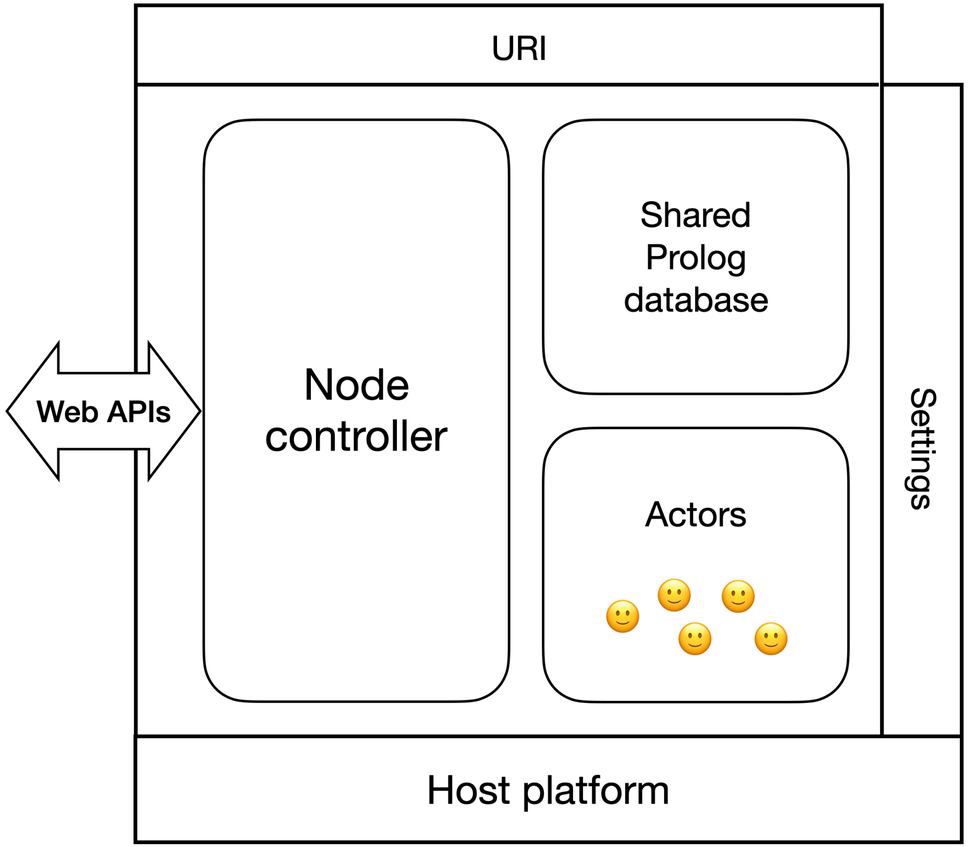
Webizing Prolog

1. Introduce URIs into the ecosystem,
2. exploit the existing web infrastructure,
3. make use of existing means for security,
4. ens
5.
6. ens
7. aim for standardization.

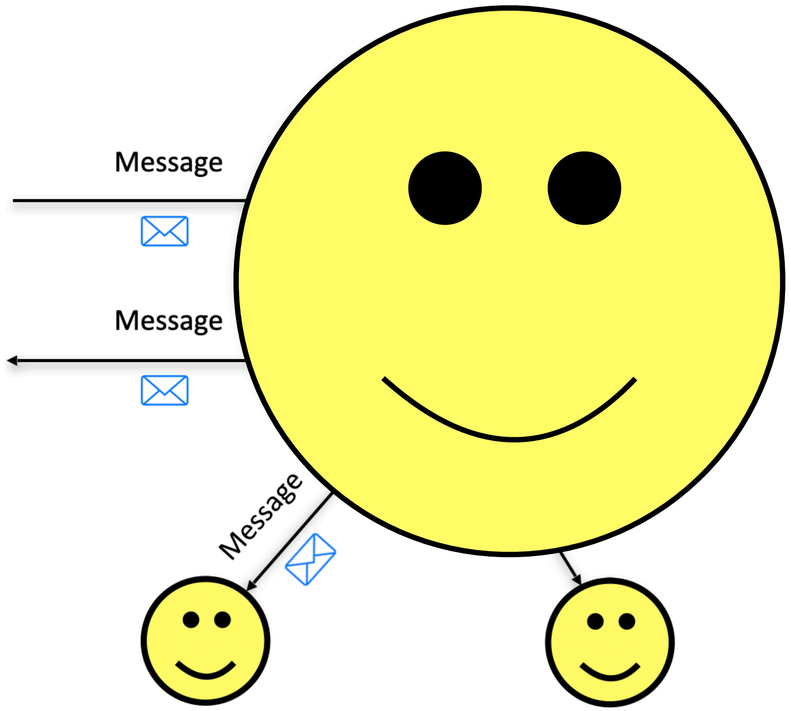
The essential process in webizing is to take a system which is designed as a closed world, and then ask what happens when it is considered as part of an open world.



An ACTOR node



The anatomy of a Prolog actor



Built-in predicates

- `spawn/2-3`
- `!/2 (or send/2)`
- `receive/1-2`
- `...`

One built-in message

- `down(Pid, Term)`

Scripting an actor that can keep a count

```
count_actor(Count0) :-
    receive({
        count(From) ->
            Count is Count0 + 1,
            From ! count(Count),
            count_actor(Count) ;
        stop ->
            true
    }).
```

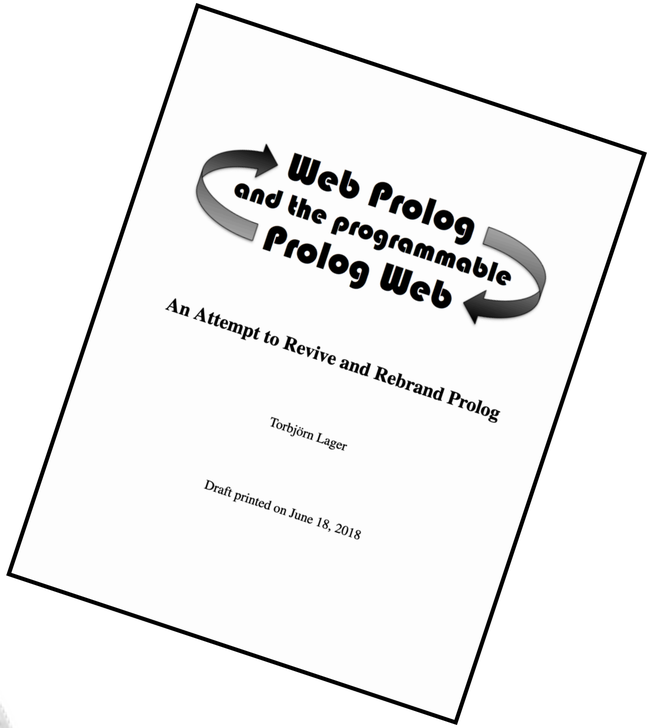
```
?- self(Self).
Self = 98732093.

?- spawn(count_actor(0), Pid, [
       monitor(true)
   ]).
Pid = 69774322.

?- $Pid ! count($Self).
true.

?- receive({Answer -> true}).
Answer = count(1).

?- $Pid ! count($Self).
true.

?- $Pid ! stop.
true.

?- flush.
Shell got count(2)
Shell got down(69774322,true)
true.

?-
```

```
?- spawn(count_actor(0), Pid, [
       monitor(true),
       node('http://n1.org'),
       load_text("
           count_actor(Count0) :-
               receive({
                   count(From) ->
                       Count is Count0 + 1,
                       From ! count(Count),
                       count_actor(Count) ;
                   stop ->
                       true
               }).
       ")
   ]).
Pid = 45092311@'http://n1.org'.

?-
```

Two actors playing ping-pong

```
ping(0, Pong_Pid) :-
    Pong_Pid ! finished,
    format('Ping finished').
ping(N, Pong_Pid) :-
    self(Self),
    Pong_Pid ! ping(Self),
    receive({
        pong ->
            format('Ping received pong')
    }),
    N1 is N - 1,
    ping(N1, Pong_Pid).

pong :-
    receive({
        finished ->
            format('Pong finished') ;
        ping(Ping_Pid) ->
            format('Pong received ping'),
            Ping_Pid ! pong,
            pong
    }).

start :-
    spawn(pong, Pong_Pid),
    spawn(ping(3, Pong_Pid), _).
```

```
?- start.
true.
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Ping finished
Pong finished

?-
```



Reading the code was fun — I had to do a double take — was I reading Erlang or Prolog — they often look pretty much the same.*

*Joe Armstrong (p.c. June 18, 2018)

Two actors playing ping-pong

Web Prolog

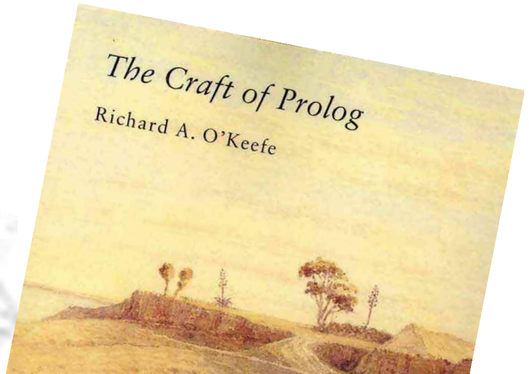
```
ping(0, Pong_Pid) :-
    Pong_Pid ! finished,
    format('Ping finished').
ping(N, Pong_Pid) :-
    self(Self),
    Pong_Pid ! ping(Self),
    receive({
        pong ->
            format('Ping received pong')
    }),
    N1 is N - 1,
    ping(N1, Pong_Pid).

pong :-
    receive({
        finished ->
            format('Pong finished’);
        ping(Ping_Pid) ->
            format('Pong received ping'),
            Ping_Pid ! pong,
            pong
    }).

start :-
    spawn(pong, Pong_Pid),
    spawn(ping(3, Pong_Pid), _).
```

Erlang

```
-module(tut15).
-export([start/0, ping/2, pong/0]).

ping(0, Pong_Pid) ->
    Pong_Pid ! finished,
    io:format('Ping finished');
ping(N, Pong_Pid) ->
    Pong_Pid ! {ping, self()},
    receive
        pong ->
            io:format('Ping received pong')
    end,
    ping(N - 1, Pong_Pid).

pong() ->
    receive
        finished ->
            io:format('Pong finished');
        {ping, Ping_Pid} ->
            io:format('Pong received ping'),
            Ping_Pid ! pong,
            pong()
    end.

start() ->
    Pong_Pid = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_Pid]).
```

% Example from: https://erlang.org/doc/getting_started/conc_prog.html

I would prefer multi-threading in Prolog to look as much as possible like Erlang.*



*Quote from the **erlang-programming** mailing list.

Code size comparison

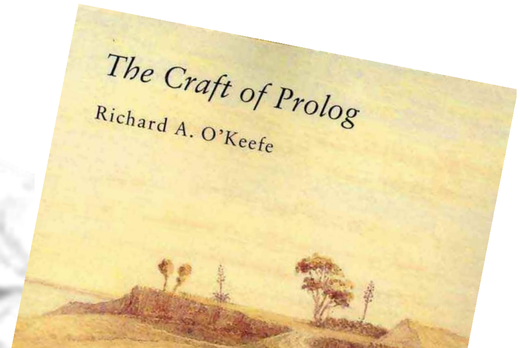
My parallel/1: 38 lines of code

```
parallel(Goals) :-
    maplist(par_call, Goals, Pids),
    maplist(par_yield(Pids), Pids, Goals).

par_call(Goal, Pid) :-
    self(Self),
    spawn((call(Goal), Self ! Pid-Goal), Pid, [
        monitor(true)
    ]).

par_yield(Pids, Pid, Goal) :-
    receive({
        down(Pid, true) ->
            true ;
        down(_, false) ->
            tidy_up_all(Pids),
            !, fail ;
        down(_, exception(E)) ->
            tidy_up_all(Pids),
            throw(E)
    }),
    receive({Pid-Goal -> true}).

tidy_up_all(Pids) :-
    maplist(tidy_up, Pids).

tidy_up(Pid) :-
    demonitor(Pid),
    exit(Pid, kill),
    mailbox_rm(Pid).

mailbox_rm(Pid) :-
    receive({
        Msg if arg(1, Msg, Pid) ->
            mailbox_rm(Pid)
    },[
        timeout(0)
    ]).
```

Wielemaker's concurrent/3: 101 lines of code

```
:- meta_predicate concurrent(+, :, +).

concurrent(1, M:List, _) :-
    !,
    maplist(once_in_module(M), List).
concurrent(N, M:List, Options) :-
    must_be(positive_integer, N),
    must_be(list(callable), List),
    length(List, JobCount),
    message_queue_create(Done),
    message_queue_create(Queue),
    WorkerCount is min(N, JobCount),
    create_workers(WorkerCount, Queue, Done, Workers, Options),
    submit_goals(List, 1, M, Queue, VarList),
    forall(between(1, WorkerCount, _),
           thread_send_message(Queue, done)),
    VT =.. [vars|VarList],
    concur_wait(JobCount, Done, VT, cleanup(Workers, Queue),
                Result, [], Exitted),
    subtract(Workers, Exitted, RemainingWorkers),
    concur_cleanup(Result, RemainingWorkers, [Queue, Done]),
    (   Result == true
    ->  true
    ;   Result = false
    ->  fail
    ;   Result = exception(Error)
    ->  throw(Error)
    ).

once_in_module(M, Goal) :-
    call(M:Goal), !.

submit_goals([], _, _, _, []).
submit_goals([H|T], I, M, Queue, [Vars|VT]) :-
    term_variables(H, Vars),
    thread_send_message(Queue, goal(I, M:H, Vars)),
    I2 is I + 1,
    submit_goals(T, I2, M, Queue, VT).

concur_wait(0, _, _, _, true, Exited, Exited) :- !.
concur_wait(N, Done, VT, Cleanup, Status, Exitted0, Exitted) :-
    catch(thread_get_message(Done, Exit), Error,
          concur_abort(Error, Cleanup, Done, Exitted0)),
    (   Exit = done(Id, Vars)
    ->  arg(Id, VT, Vars),
        N2 is N - 1,
        concur_wait(N2, Done, VT, Cleanup, Status, Exitted0, Exitted)
    ;   Exit = finished(Thread)
    ->  thread_join(Thread, JoinStatus),
        (   JoinStatus == true
        ->  concur_wait(N, Done, VT, Cleanup, Status, [Thread|Exitted0], Exitted)
        ;   Status = JoinStatus,
            Exitted = [Thread|Exitted0]
        )
    ).

concur_abort(Error, cleanup(Workers, Queue), Done, Exitted) :-
    subtract(Workers, Exitted, RemainingWorkers),
    concur_cleanup(Error, RemainingWorkers, [Queue, Done]),
    throw(Error).

create_workers(N, Queue, Done, [Id|Ids], Options) :-
    N > 0,
    !,
    thread_create(worker(Queue, Done), Id,
                  [ at_exit(thread_send_message(Done, finished(Id)))
                  | Options
                  ]),
    N2 is N - 1,
    create_workers(N2, Queue, Done, Ids, Options).
create_workers(_, _, _, [], _).

worker(Queue, Done) :-
    thread_get_message(Queue, Message),
    (   Message = goal(Id, Goal, Vars)
    ->  (   Goal
        ->  thread_send_message(Done, done(Id, Vars)),
            worker(Queue, Done)
        )
    ;   true
    ).

concur_cleanup(Result, Workers, Queues) :-
    !,
    (   Result == true
    ->  true
    ;   kill_workers(Workers)
    ),
    join_all(Workers),
    maplist(message_queue_destroy, Queues).

kill_workers([]).
kill_workers([Id|T]) :-
    catch(thread_signal(Id, abort), _, true),
    kill_workers(T).

join_all([]).
join_all([Id|T]) :-
    thread_join(Id, _),
    join_all(T).
```

I would argue that it is precisely the 'receive' construct in Erlang that makes Erlang such a joy to use.*



*Quote from the **erlang-programming** mailing list.

More about receive/1-2

- Predicate: `receive/1-2`:

```
receive(+ReceiveClauses) is semidet.
receive(+ReceiveClauses, :Options) is semidet.
```

Where `ReceiveClauses` takes the form:

```
{ Pattern1 [if Guard1] ->
      Body1 ;
  ...
  PatternN [if GuardN] ->
      BodyN
}
```

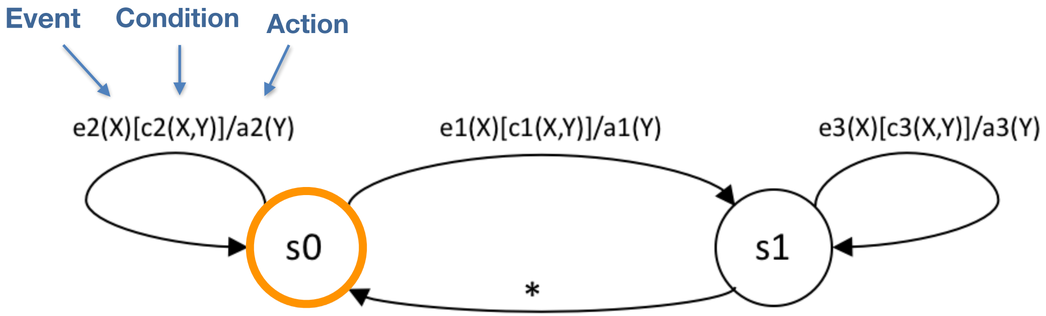
Valid `Options`:

```
timeout(IntegerOrFloat)
on_timeout(Goal)
```

```
ask_name_and_age :-
    format('What is your name?~n'),
    receive({
        name(Name) ->
            format("Hello ~w.~n", [Name]),
            format('How old are you?~n'),
            receive({
                age(Age) if Age < 20 ->
                    format('You are too young!~n') ;
                age(Age) if Age >= 20 ->
                    format('You are old enough.~n')
            })
    },[ timeout(10),
        on_timeout(ask_name_and_age)
    ]).
```

```
?- spawn(ask_name_and_age, Pid).
Pid = 16078857.

What is your name?

What is your name?

?- $Pid ! name('Joe').
true.

Hello Joe.
How old are you?

?- $Pid ! age(19).
You are too young!
true.

?-
```

An event-driven state machine




```
s0 :-
    format("Entered s0.~n"),
    receive({
        e1(X) if c1(X,Y) ->
            a1(Y),
            s1 ;
        e2(X) if c2(X,Y) ->
            a2(Y),
            s0
    }).
```

```
s1 :-
    format("Entered s1.~n"),
    receive({
        e2(X) if c3(X,Y) ->
            a3(Y),
            s1 ;
        _AnyEvent  ->
            s0
    }).
```

```
?- spawn(s0, Pid).
Entered s0.
Pid = 17746747.

?- $Pid ! e1(a).
Action a1(b) executed.
Entered s1.

?- $Pid ! e4(c).
Entered s0.

?-
```

```
c1(a,b).

a1(A) :-
    format("Action a1(~p) executed.~n",[A]).
```

This kind of guard is not permitted in Erlang!

Receive is semi-deterministic

- Predicate: `receive/1-2:`

```
receive(+ReceiveClauses) is semidet.
receive(+ReceiveClauses, :Options) is semidet.
```

- `receive/1-2` is *semi-deterministic*, i.e. it either fails, or succeeds exactly once.

- The only way it will fail is if the goal in the body of one of its receive clauses fails, or if a timeout occurs and the goal passed in the `on_timeout` option fails.

```
ping_server :-
    receive({
        ping(Pid) ->
            Pid ! pong,
            ping_server
    }).
```

```
ping_server :-
    repeat,
    receive({
        ping(Pid) ->
            Pid ! pong,
            fail
    }).
```

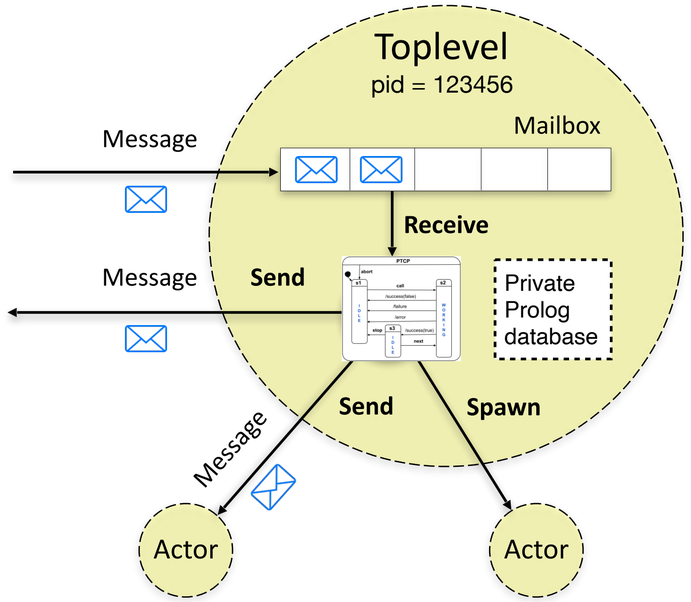
A simple toplevel actor

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

toplevel(Pid, Options) :-
    self(Self),
    spawn(session(Pid, Self), Pid, Options).

session(Pid, Parent) :-
    receive({
        '$call'(Template, Goal) ->
            (   call_cleanup(Goal, De
                (   var(Det)
                ->  Parent ! success(Pid,
                    receive({
                        '$next' -> fail ;
                        '$stop' -> true
                    })
                ;   Parent ! success(Pid, Template, false)
                )
            ;   Parent ! failure(Pid)
            )
    }),
    session(Pid, Parent).



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

toplevel_next(Pid) :-
    Pid ! '$next'.

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

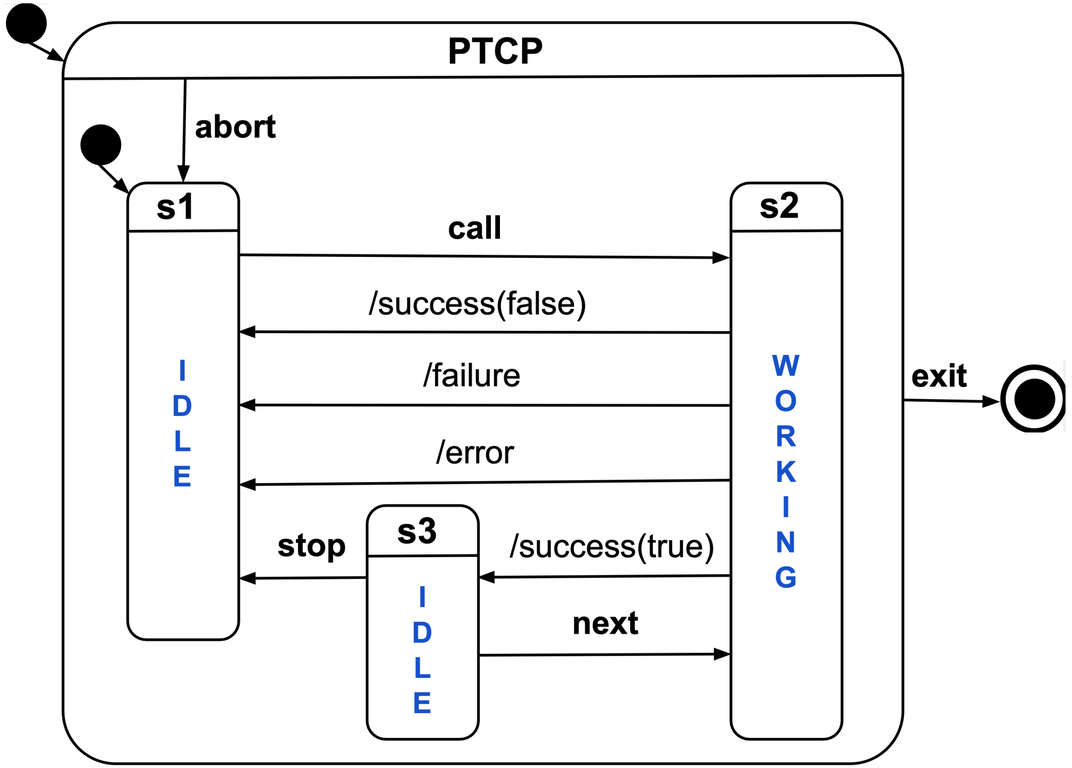
No need for this!

```
?- toplevel(Pid, [
       monitor(true)
   ]).
Pid = 12929949.

?- toplevel_call($Pid, I, between(1,3,I)).
true.

?- flush.
Shell got success(12929949,1,true)
true.

?- toplevel_next($Pid).
true.

?- flush.
l got success(12929949,2,true)

evel_next($Pid).
true.

?- flush.
Shell got success(12929949,3,false)
true.

?- exit($Pid, killed).
true.

?- flush.
Shell got down(12929949,killed)
true.

?-
```

A toplevel is an actor with a built-in protocol



The Prolog Toplevel Communication Protocol



Built-in predicates

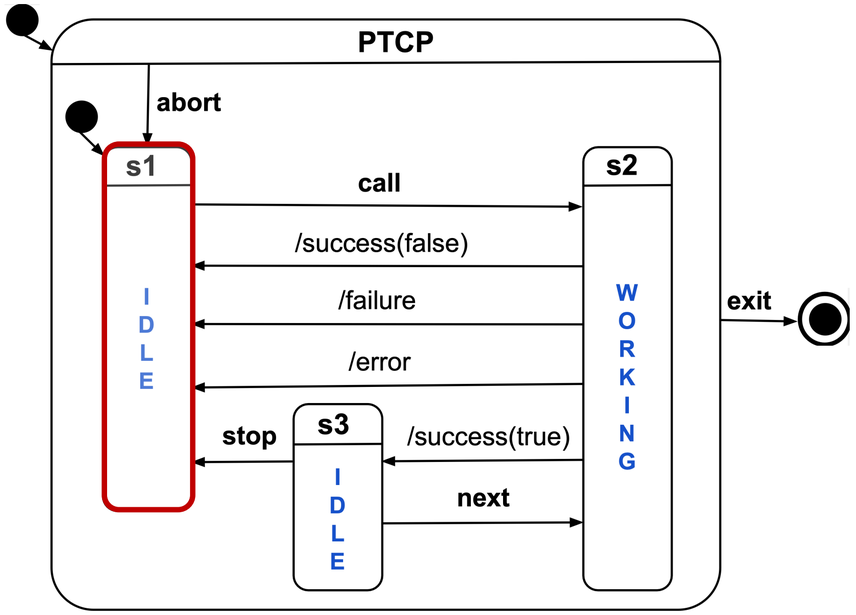
- `toplevel_spawn/1-2`
- `toplevel_call/2-3`
- `toplevel_next/1-2`
- `toplevel_stop/1-2`
- `toplevel_abort/1`
- `toplevel_exit/1`

Built-in messages

- `success(Pid, [Terms], Bool)`
- `failure(Pid)`
- `error(Pid, Term)`

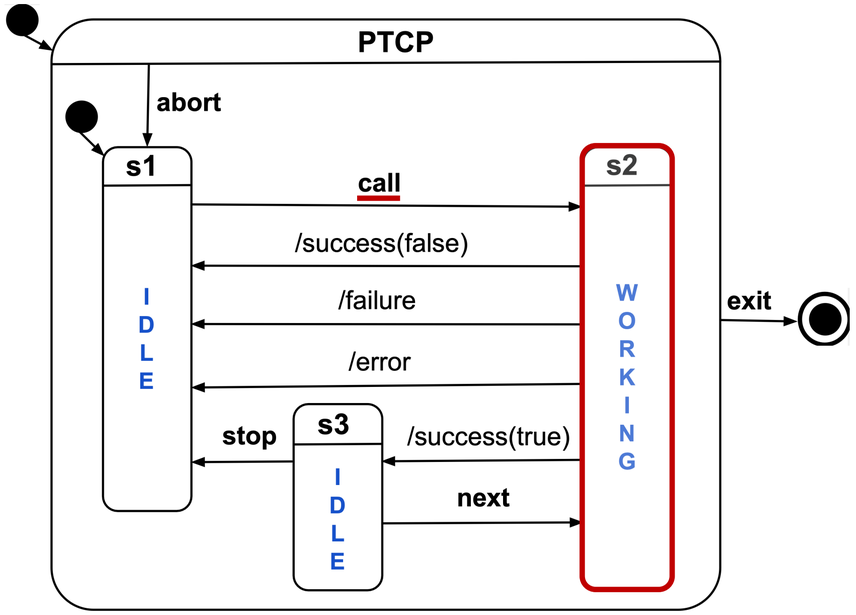
A session with a toplevel actor

```
?- toplevel_spawn(Pid, [
       monitor(true)
   ]).
Pid = 75287290.

?-
```



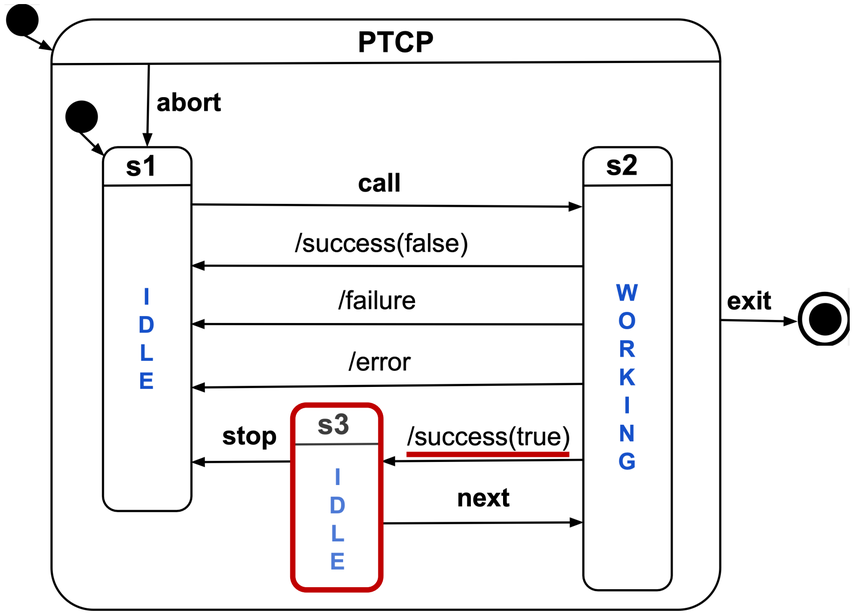
A session with a toplevel actor

```
?- toplevel_spawn(Pid, [
      monitor(true)
   ]).
Pid = 75287290.

?- toplevel_call($Pid, p(X), [
       limit(1),
       template(X)
   ]).
true.

?-
```



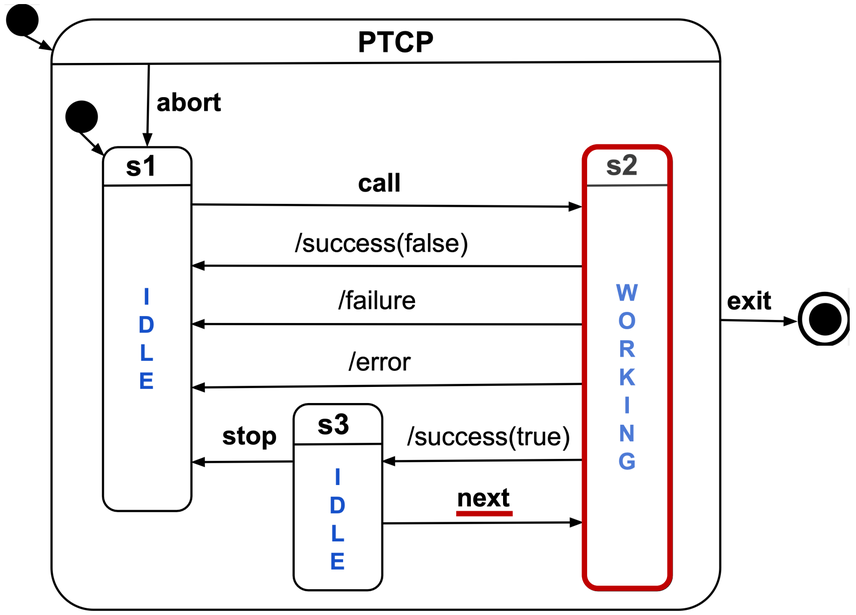
A session with a toplevel actor

```
?- toplevel_spawn(Pid, [
      monitor(true)
   ]).
Pid = 75287290.

?- toplevel_call($Pid, p(X), [
       limit(1),
       template(X)
   ]).
true.

?- flush.
Shell got success(75287290,[a], true)
true.

?-
```



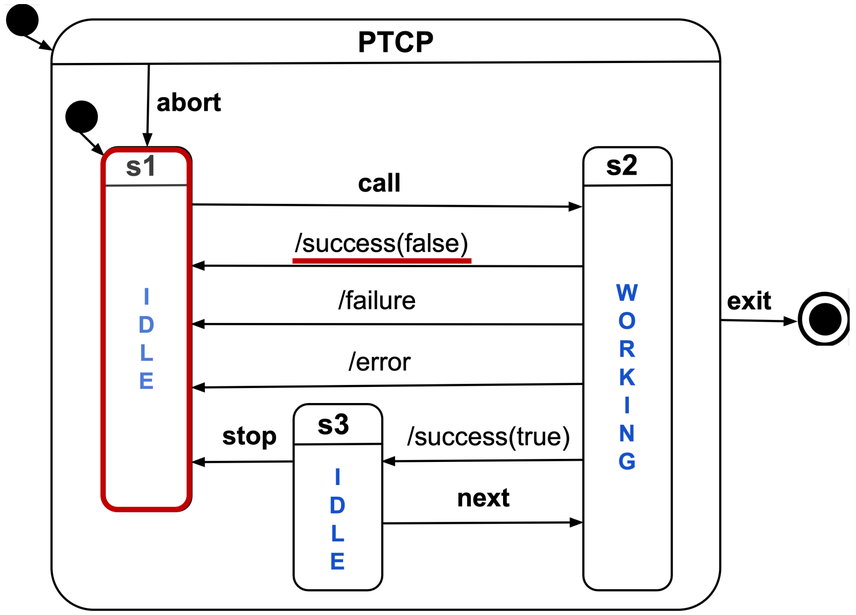
A session with a toplevel actor

```
?- toplevel_spawn(Pid, [
      monitor(true)
   ]).
Pid = 75287290.

?- toplevel_call($Pid, p(X), [
       limit(1),
       template(X)
   ]).
true.

?- flush.
Shell got success(75287290,[a], true)
true.

?- toplevel_next($Pid, [
       limit(10)
   ]).
true.

?-
```



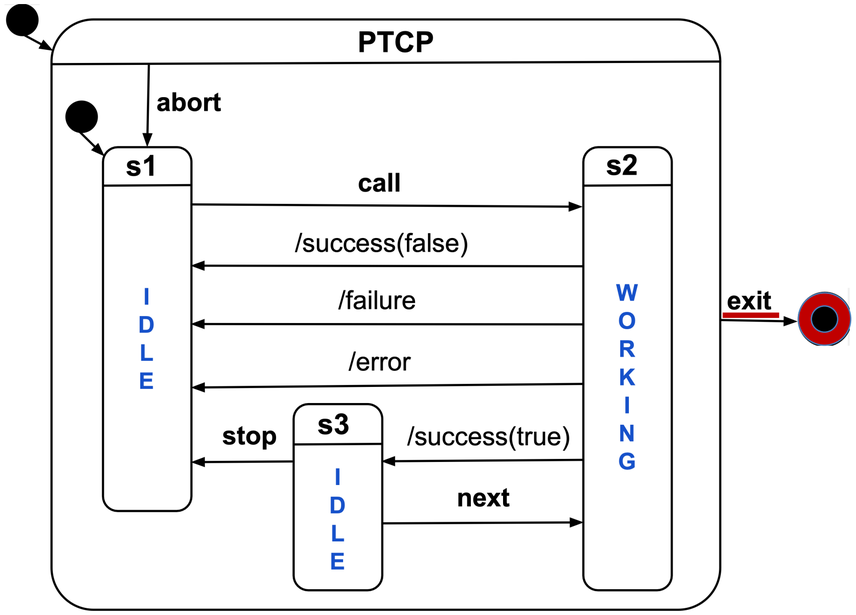
A session with a toplevel actor

```
?- toplevel_spawn(Pid, [
      monitor(true)
   ]).
Pid = 75287290.

?- toplevel_call($Pid, p(X), [
       limit(1),
       template(X)
   ]).
true.

?- flush.
Shell got success(75287290,[a], true)
true.

?- toplevel_next($Pid, [
       limit(10)
   ]).
true.

?- flush.
Shell got success(75287290,[b,c], false)
true.

?-
```



A session with a toplevel actor

```
?- toplevel_spawn(Pid, [
      monitor(true)
   ]).
Pid = 75287290.

?- toplevel_call($Pid, p(X), [
       limit(1),
       template(X)
   ]).
true.

?- flush.
Shell got success(75287290,[a], true)
true.

?- toplevel_next($Pid, [
       limit(10)
   ]).
true.

?- flush.
Shell got success(75287290,[b,c], false)
true.

?- exit($Pid, goodbye).
true.

?- flush.
Shell got down(75287290, goodbye)
true.
```

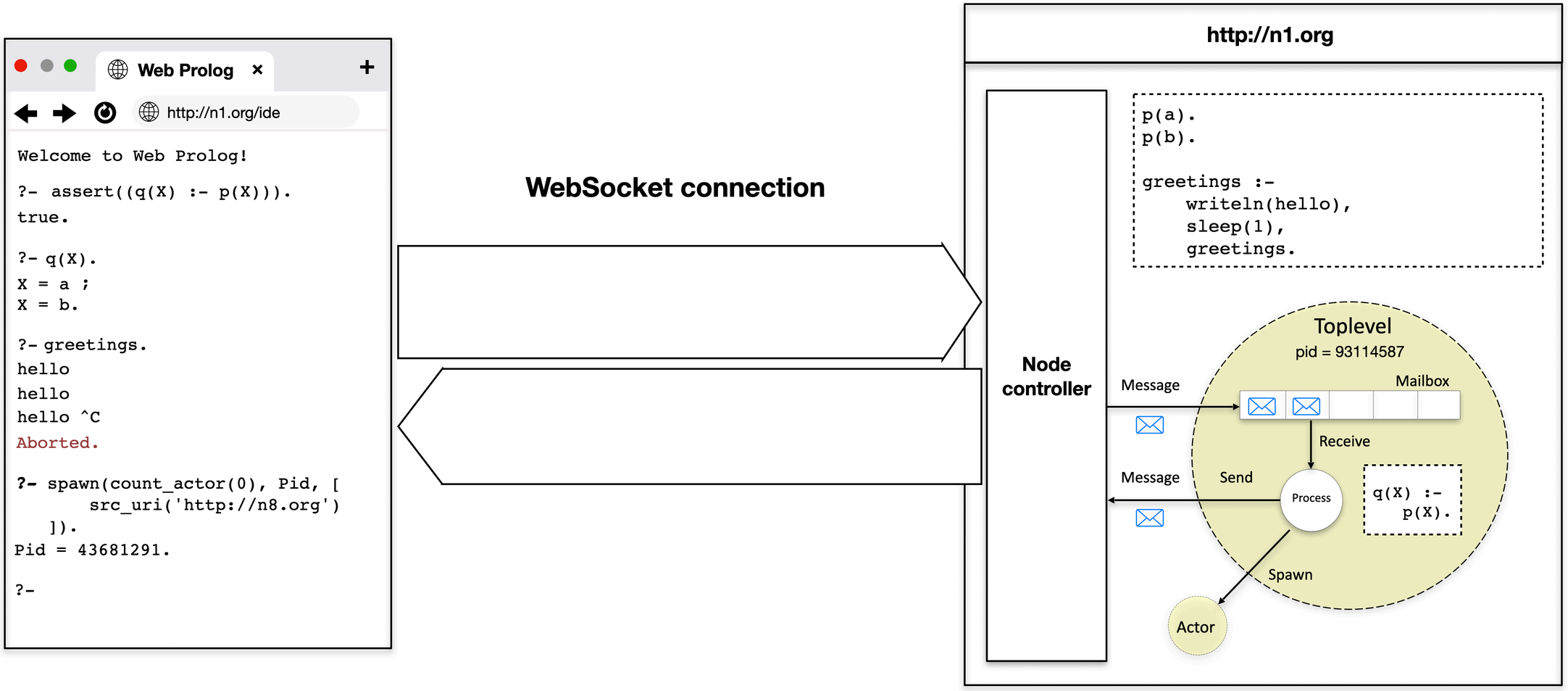


An implementation of rpc/2-3 on top of a toplevel

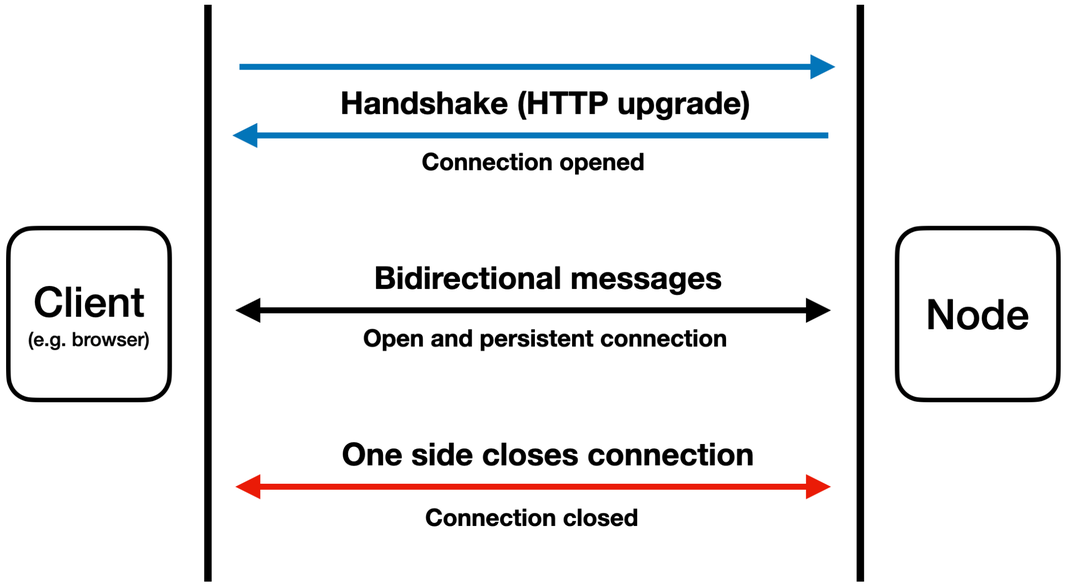
```
rpc(URI, Goal) :-
    rpc(URI, Goal, []).

rpc(URI, Goal, Options) :-
    toplevel_spawn(Pid, [
         node(URI),
         session(false)
       | Options
    ]),
    toplevel_call(Pid, Goal, Options),
    wait_answer(Pid, Goal).

wait_answer(Pid, Goal) :-
    receive({
        success(Pid, Slice, true) ->
            (   member(Goal, Slice)
            ;   toplevel_next(Pid),
                wait_answer(Pid, Goal)
            ) ;
        success(Pid, Slice, false) ->
            member(Goal, Slice) ;
        failure(Pid) -> !, fail ;
        error(Pid, Error) ->
            throw(Error)
    }).
```

```
?- rpc('http://n1.org', human(Who)).
Who = plato ;
Who = aristotle.
?-

?- rpc('http://n1.org', human(Who), [
        limit(1)
   ]).
Who = plato ;
Who = aristotle.
?-

?- rpc('http://n1.org', mortal(Who), [
        src_text("
            mortal(Who) :-
                human(Who).
        ")
   ]).
Who = plato ;
Who = aristotle.
?-
```

Browser talking to a toplevel actor

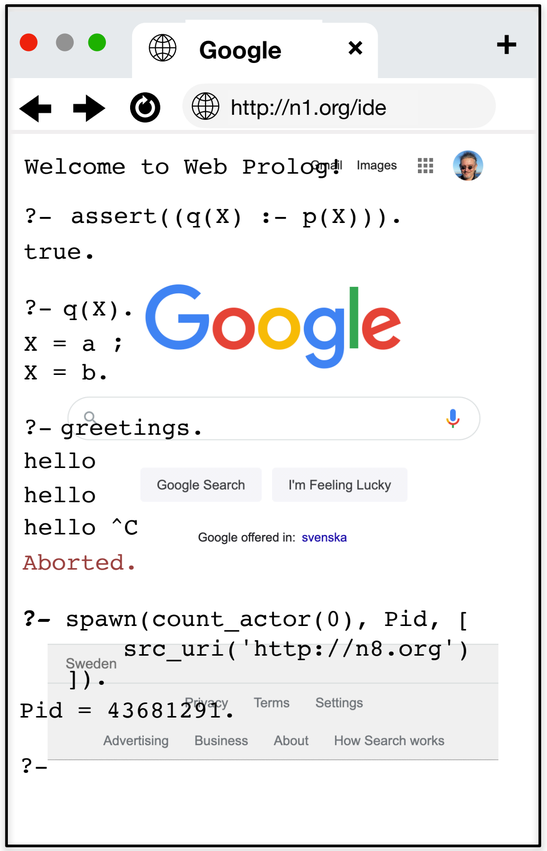


The stateful WebSocket API

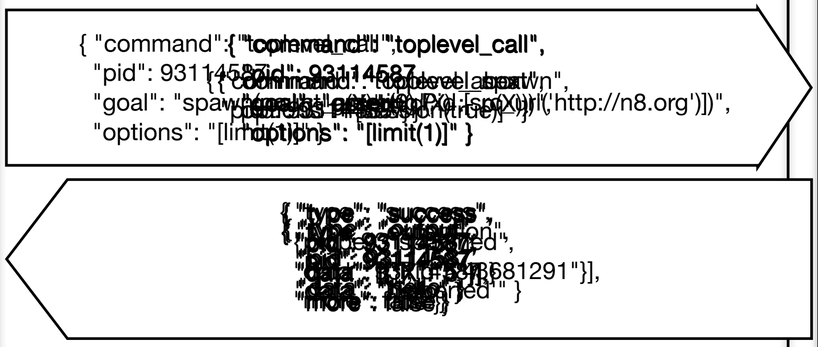


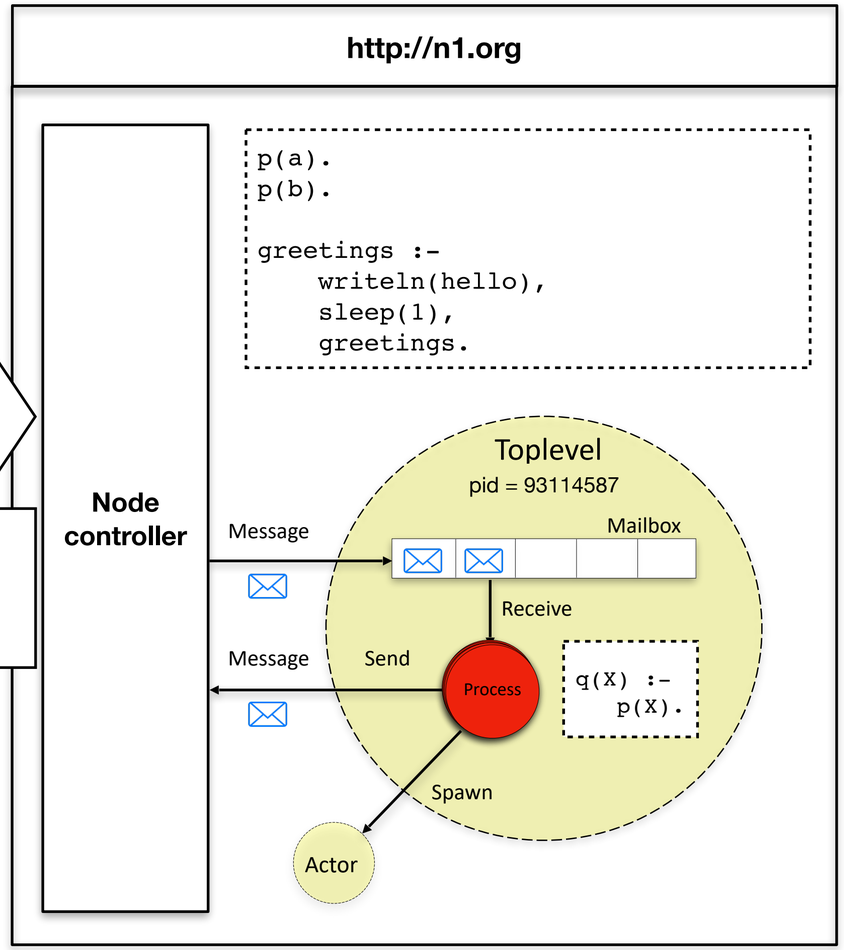
- WebSocket is a real-time, low latency, bi-directional protocol for asynchronous communication between a client and a server.
- A WebSocket sub-protocol that uses JSON messages has been defined.
- Can be used for client-to-node communication as well as for node-to-node communication.

Browser talking to a toplevel actor



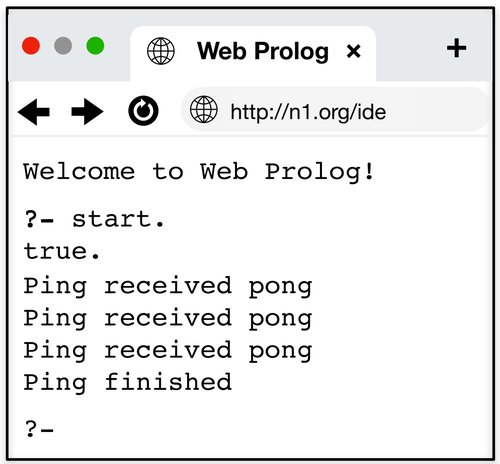
WebSocket connection

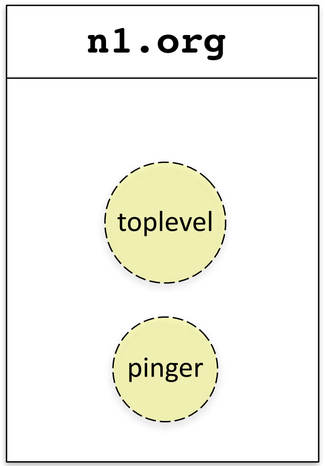


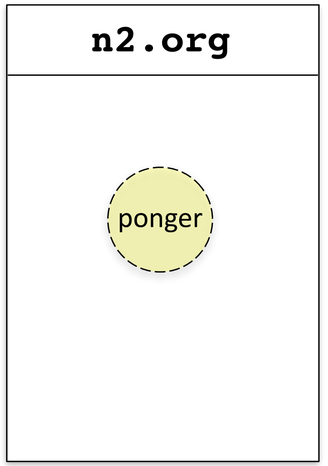


```
ping(0, Pong_Pid) :-
    Pong_Pid ! finished,
    format('Ping finished').
ping(N, Pong_Pid) :-
    self(Self),
    Pong_Pid ! ping(Self),
    receive({
        pong ->
            format('Ping received pong')
    }),
    N1 is N - 1,
    ping(N1, Pong_Pid).

start :-
    spawn(pong, Pong_Pid, [
        node('http://n2.org')
    ]),
    spawn(ping(3, Pong_Pid), _).
```

```
pong :-
    receive({
        finished ->
            format('Pong finished');
        ping(Ping_Pid) ->
            format('Pong received ping'),
            Ping_Pid ! pong,
            pong
    }).
```







The stateful API — pros and cons

Pros

- Clients can have almost full control of their own Web Prolog processes.
- Full Prolog, including constructs with side effects such as I/O and database manipulation, can be supported.
- Even predicates for spawning processes and sending and receiving messages can be supported.

Cons

- Clients tend to tie up parts of a node's resources for as long as a session lasts.
- Scalability might be a problem.

Access would have to be restricted

Implementing the ACTOR profile...

... on top of the ISO Prolog Threads (draft) standard

To implement these:

- spawn(+Goal, -Pid, +Options)
- self(?Pid)
- +Pid ! +Message
- receive(+ReceiveClauses, +Options)
- exit(+Pid, +Reason)

... you would need these:

- thread_create(+Goal, -ID, +Options)
- thread_self(?ID)
- thread_send_message(+ID, +Message)
- thread_get_message(+ID, ?Message, +Options)
- thread_signal(+ID, +Goal)
- thread_detach(+ID)
- thread_property(?ID, ?Property)

- thread_local/1 (directive)

... but you won't need these:

- message_queue_create(-Queue)
- message_queue_create(-Queue, +Options)
- message_queue_destroy(+Queue)

- mutex_create(?Mutex)
- mutex_create(-Mutex, ++Options)
- mutex_destroy(+Mutex)

- thread_cancel(+Thread)
- thread_default(?Option)
- thread_exit(+Term)
- thread_join(+Thread, -Status)
- thread_peek_message(-Message)
- thread_peek_message(+Queue, -Message)
- thread_set_default(++Option)
- thread_sleep(+Seconds)

- is_thread(?Term)
- with_mutex(+Mutex, +Goal)

More on this in Add-on C!

Maybe you shouldn't do it in this way?

WASM/WASI ?

Building and deploying applications

Theory and Practice of Logic Programming

Fifty Years of Prolog and Beyond *

PHILIPP KÖRNER, MICHAEL LEUSCHEL
Institut für Informatik, Universität Düsseldorf, Universitätsstr. 1, D-40225 Düsseldorf
(*e-mail:* {p.koerner,leuschel}@uni-duesseldorf.de)

JOÃO BARBOSA, VÍTOR SANTOS COSTA
Department of Computer Science, Faculty of Science of the University of Porto
(*e-mail:* {joao.barbosa,vscosta}@fc.up.pt)

VERÓNICA DAHL
Computing Sciences Department, Simon Fraser University
(*e-mail:* veronica_dahl@sfu.ca)

MANUEL V. HERMENEGILDO, JOSE F. MORALES
IMDEA Software Institute and Universidad Politécnica de Madrid (UPM)
(*e-mail:* {manuel.hermenegildo,josef.morales}@imdea.org)

JAN WIELEMAKER
Centrum voor Wiskunder en Informatica (CWI), Amsterdam
(*e-mail:* J.Wielemaker@cwi.nl)

DANIEL DIAZ
Centre de Recherche en Informatique, University Paris-1
(*e-mail:* daniel.diaz@univ-paris1.fr)

SALVADOR ABREU
NOVA-LINCS, University of Évora
(*e-mail:* spa@uevora.pt)

GIOVANNI CIATTO
Dept. of Computer Science and Engineering, Alma Mater Studiorum—Univerità di Bologna
(*e-mail:* giovanni.ciatto@unibo.it)

Table 3. SWOT Analysis

Strengths (Section 4.1)

- clean, simple language syntax and semantics
- immutable persistent data structures, with "declarative" pointers (logic variables)
- arbitrary precision arithmetic
- safety (garbage collection, no NullPointer exceptions, ...)
- tail-recursion and last-call optimization
- efficient inference, pattern matching, and unification, DCGs
- meta-programming, programs as data
- constraint solving (3.3.2), independence of the selection rule (coroutines (3.3.3))
- indexing (3.3.3), efficient tabling (3.3.3)
- fast development, REPL (Read, Execute, Print, Loop), debugging (3.3.4)
- commercial (2.10.1) and open-source systems
- active developer community with constant new implementations, features, etc.
- sophisticated tools: analyzers, partial evaluators, parallelizers, ...
- successful applications
  - program analysis,
  - domain-specific languages
  - heterogeneous data integration
  - natural language processing
  - efficient inference (expert systems, theorem provers), symbolic AI
- many books, courses and learning materials

Weaknesses (Section 4.3)

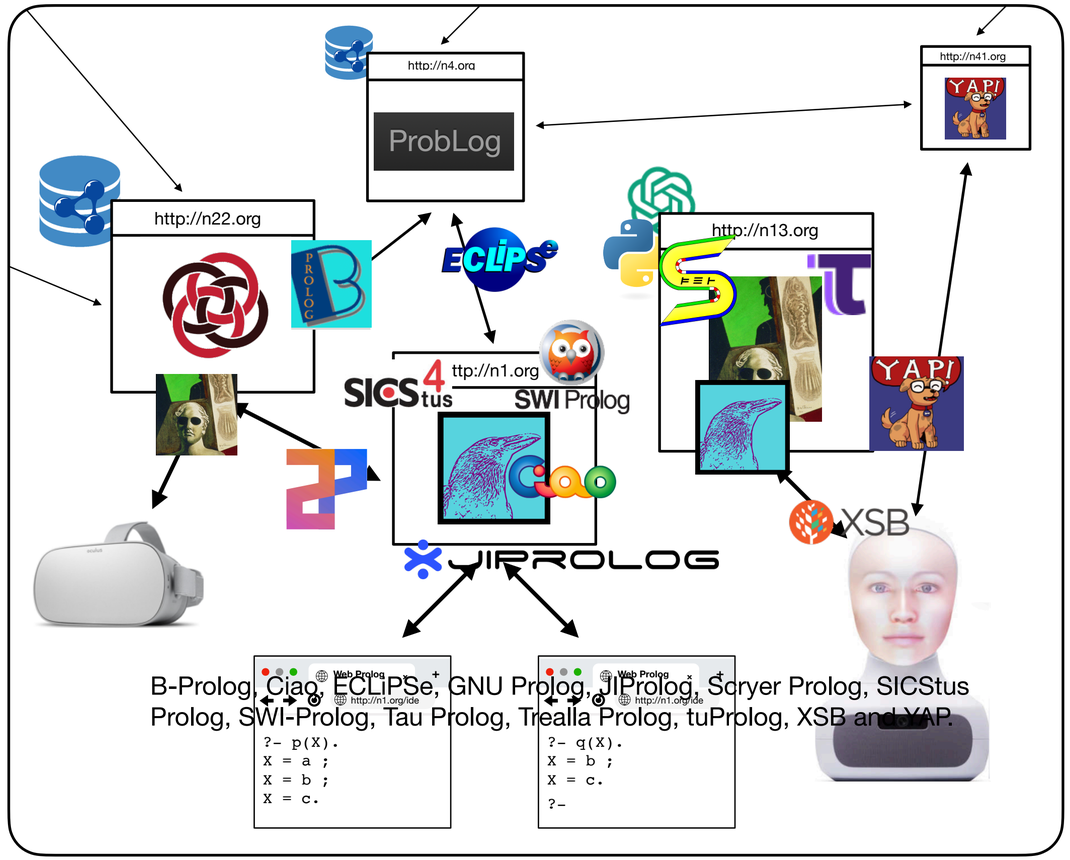
- syntactically different from "traditional" programming languages, not a mainstream language
- learning curve, beginners can easily write programs that loop or consume a huge amount of resources
- static typing (see, however, 3.3.3)
- data hiding (see, however, 3.3.1)
- object orientation (see, however, 4.5.4)
- limited portability (see 4.5.1)
- packages: availability and management
- IDEs and development tools: limited capabilities in some areas (e.g., refactoring; 4.5.2)
- UI development (usually conducted in a foreign language via FLI (3.3.1))
- limited support for embedded or app development

Opportunities (Section 4.2)

- new application areas, addressing societal challenges 4.2:
  - neuro-Symbolic AI
  - explainable AI, verifiable AI
  - Big Data
- new features and developments
  - probabilistic reasoning (3.5.1)
  - embedding ASP (3.5.1) and SAT or SMT solving
  - parallelism (2.7, 3.3.3) (resurrecting 80s, 90s research)
  - full-fledged JIT compiler

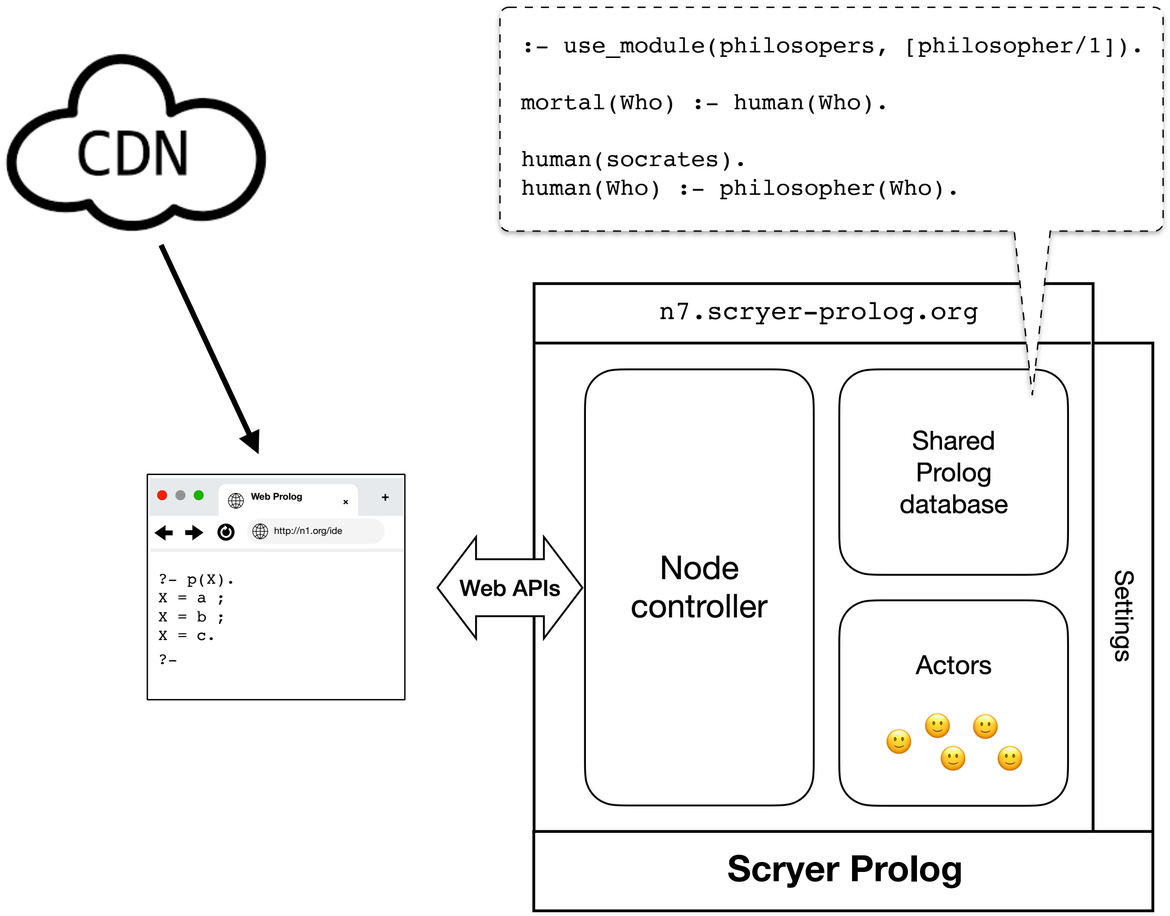
Threats (Section 4.4)

- comparatively small user base
- fragmented community with limited interactions (e.g., on StackOverflow, reddit), see 4.4.1
- active developer community with constant new implementations, features, etc.
- further fragmentation of Prolog implementations, see 4.4.1
- new programming languages
- post-desktop world of JavaScript web-applications
- the perception that it is an "old" language
- wrong image due to "shallow" teaching of the language



- *Interoperability* across implementations using Web Prolog as a *lingua franca* ACL
- JavaScript applications talking to the Prolog Web, and (for some uses) Web Prolog replacing JavaScript
- The Web as an important arena for AI
- Intelligent conversational agent as the *face* of AI
- Interfacing with big databases and RDF triple stores
- Potentially bigger user base
- A renewed, "rebranded" language!

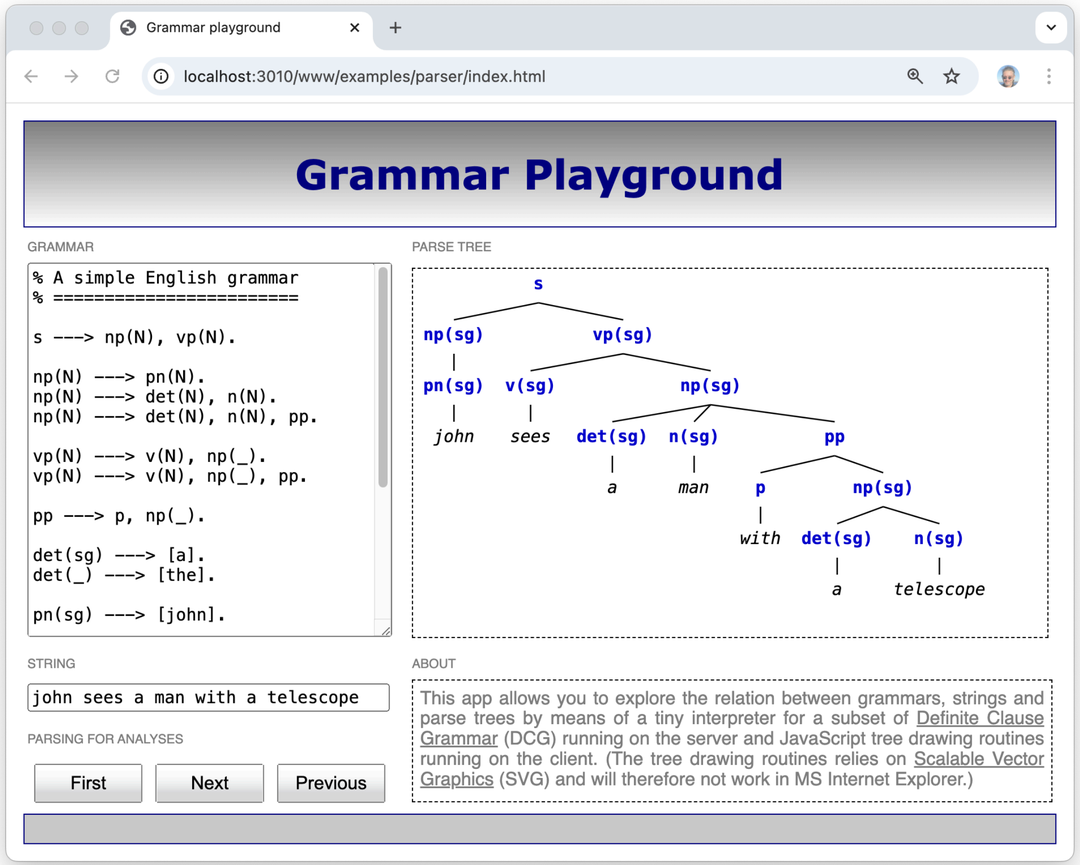
Deploying a node

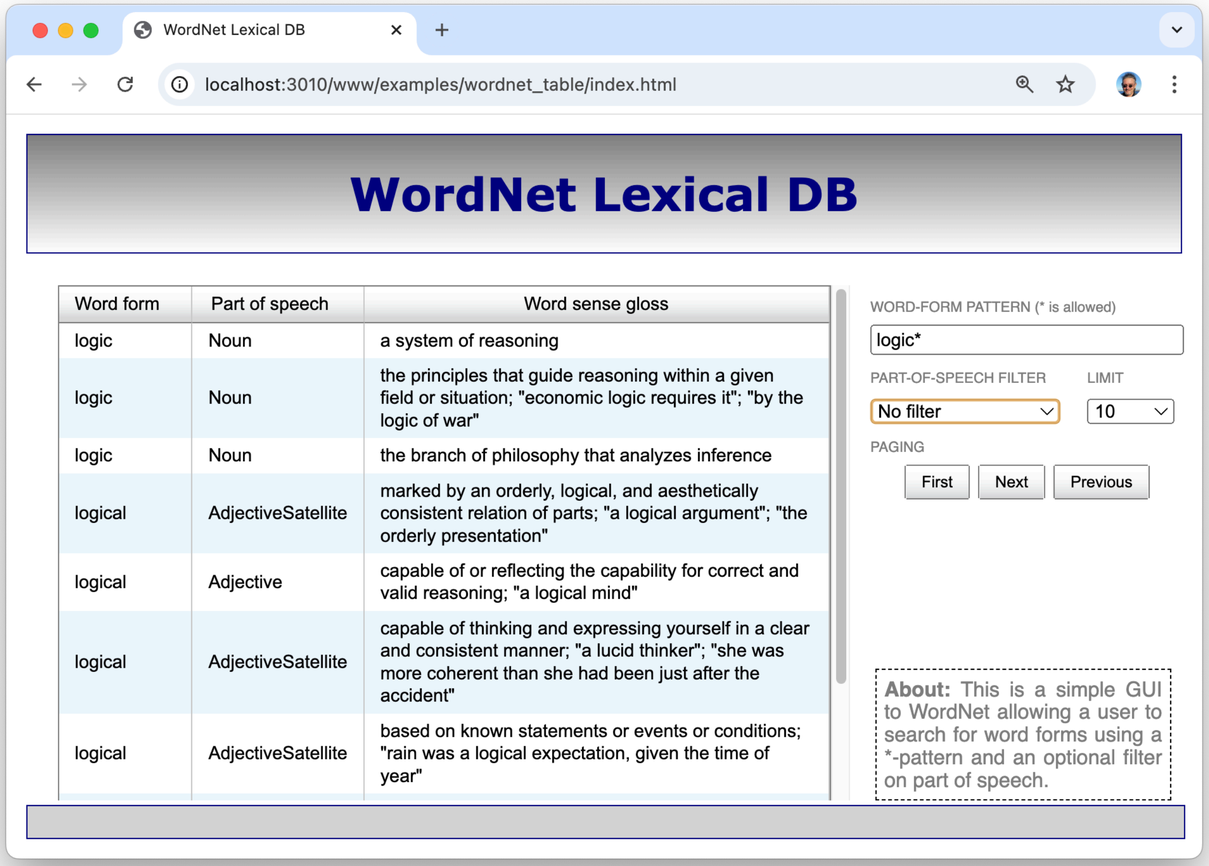


Node endpoint	Target
`https://n7.scryer-prolog.org`	The shared Prolog database
`https://n7.scryer-prolog.org/call`	The stateless HTTP API
`wss://n7.scryer-prolog.org/actor`	The stateful WebSocket API
`https://n7.scryer-prolog.org/shell`	A simple shell

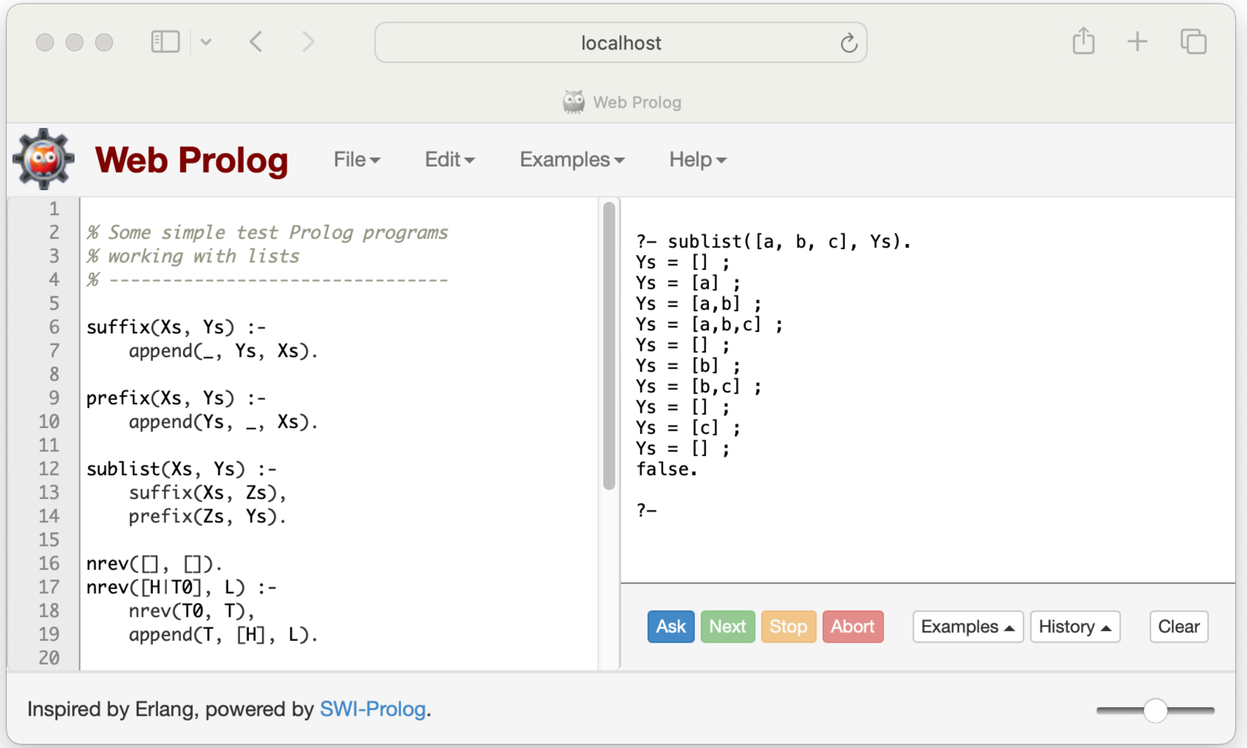
```
$ cat > n7.pl
:- use_module(philosopers, [philosopher/1]).

mortal(Who) :- human(Who).

human(socrates).
human(Who) :- philosopher(Who).
^D
$ scryer-node --port=80 --src=n7.pl --settings=s.pl

Welcome to Scryer Prolog 2.0

?-
```

Two simple applications



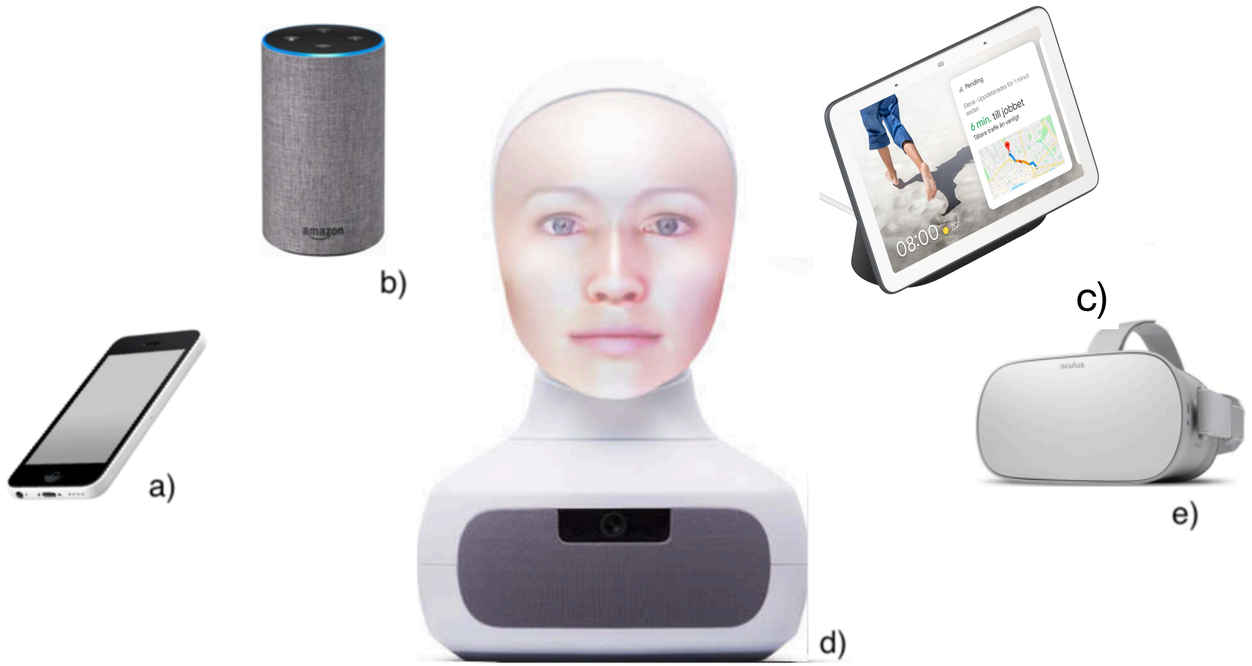


A Prolog playground

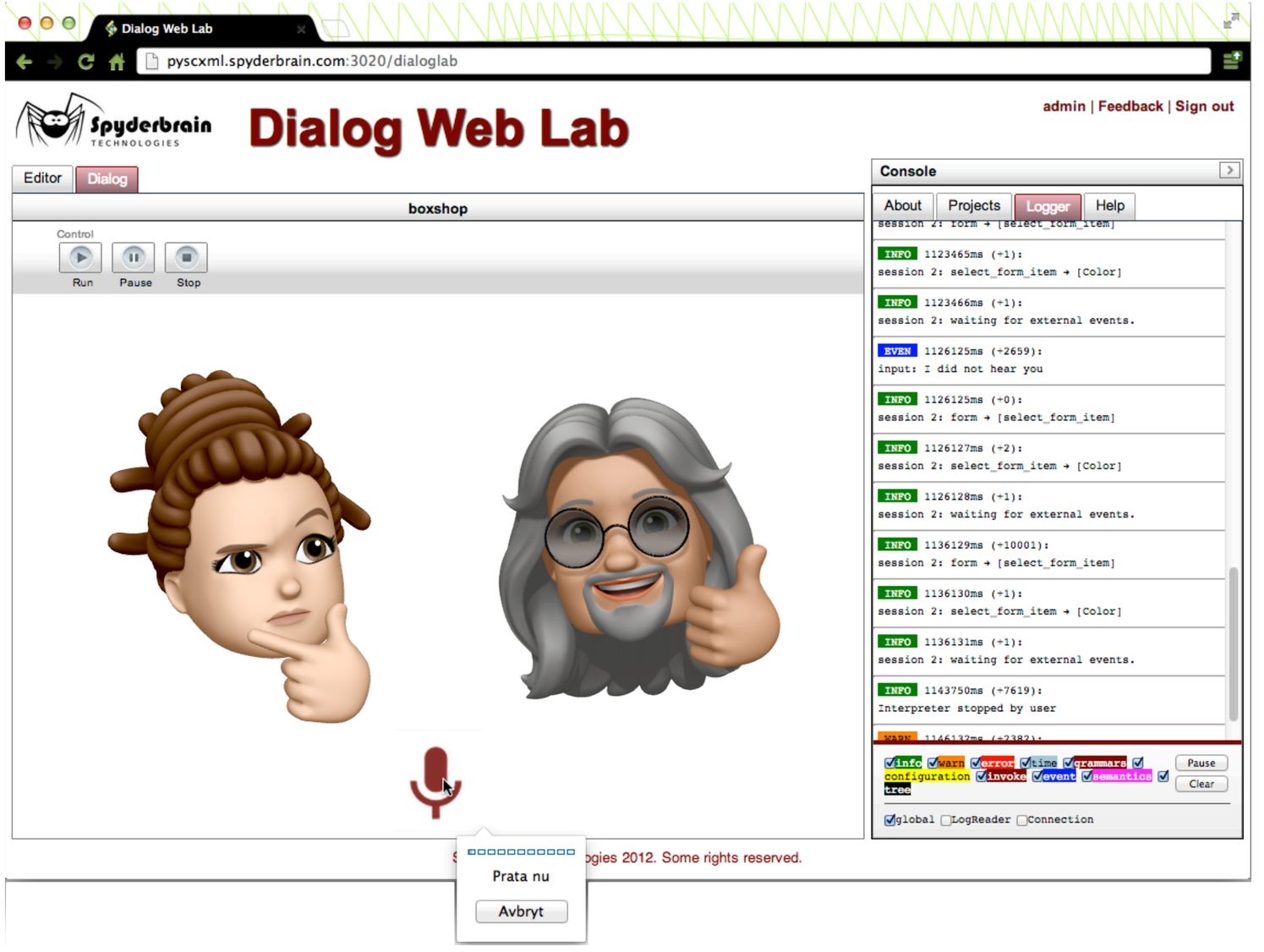



```
connection.send(JSON.stringify({
    command:"toplevel_spawn",
    options:"[load_text('" + editor.content + "')]"
}))
```

Intelligent conversational agents



They seem to need **knowledge representation** and **reasoning**, **natural language processing** as well as means for **fine-grained real-time interaction** …


Dialog Web Lab
pyscxml.spyderbrain.com:3020/dialoglab
Spyderbrain TECHNOLOGIES
Dialog Web Lab
admin | Feedback | Sign out
Editor
Dialog
boxshop
Control
Run
Pause
Stop
Console
About
Projects
Logger
Help
INFO 1123465ms (+1):
session 2: select_form_item → [Color]
INFO 1123466ms (+1):
session 2: waiting for external events.
EVEN 1126125ms (+2659):
input: I did not hear you
INFO 1126125ms (+0):
session 2: form → [select_form_item]
INFO 1126127ms (+2):
session 2: select_form_item → [Color]
INFO 1126128ms (+1):
session 2: waiting for external events.
INFO 1136129ms (+10001):
session 2: form → [select_form_item]
INFO 1136130ms (+1):
session 2: select_form_item → [Color]
INFO 1136131ms (+1):
session 2: waiting for external events.
INFO 1143750ms (+7619):
Interpreter stopped by user
info warn error time grammars configuration invoke event semantics tree
Pause
Clear
global LogReader Connection
Prata nu
Avbryt
gies 2012. Some rights reserved.

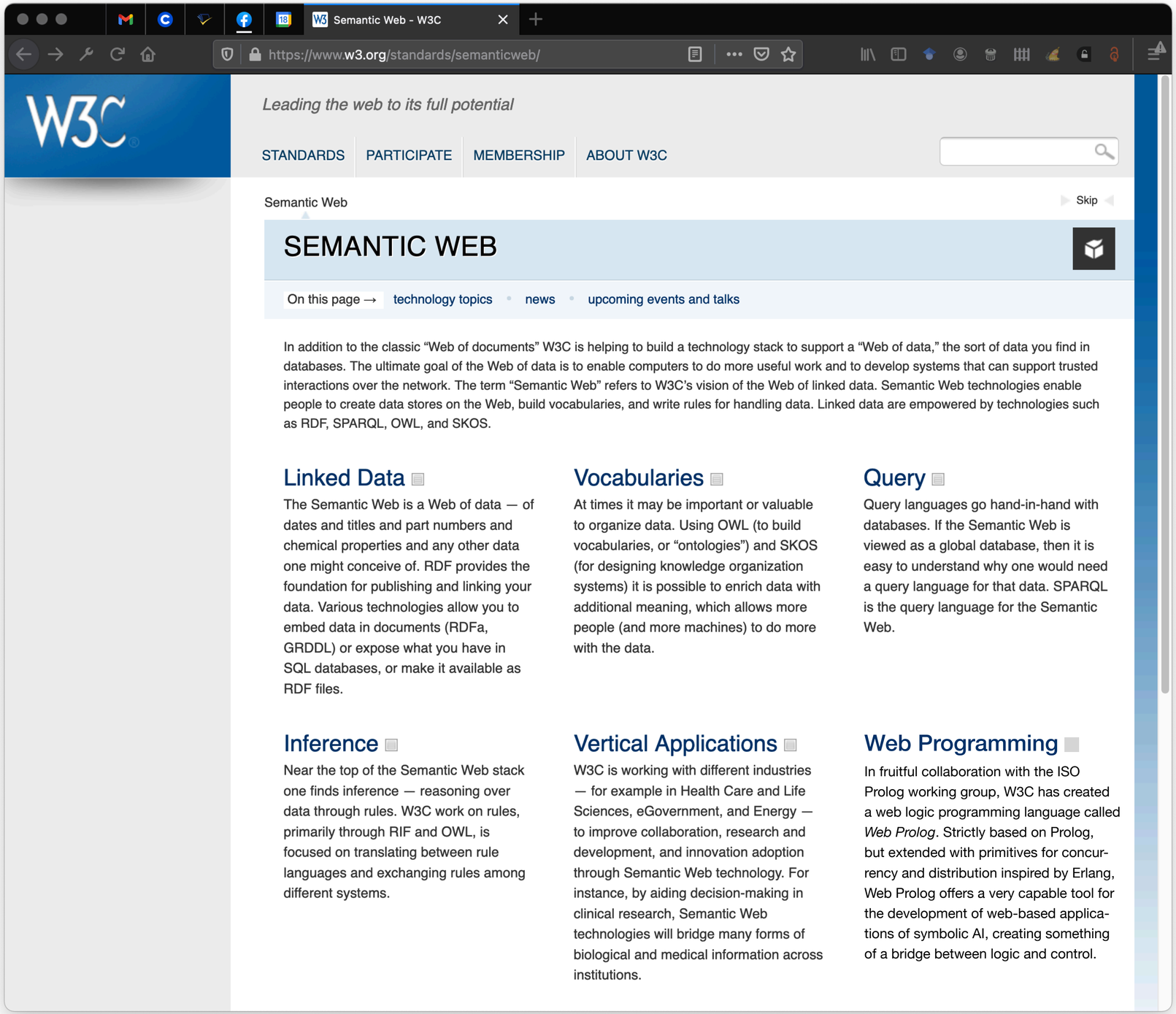
Standardizing Web Prolog — and a path to ISO Prolog 2.0

W3C Community Groups

A W3C Community Group is an open forum, without fees, where Web developers and other stakeholders develop specifications, hold discussions, develop test suites, and connect with W3C's international community of Web experts.

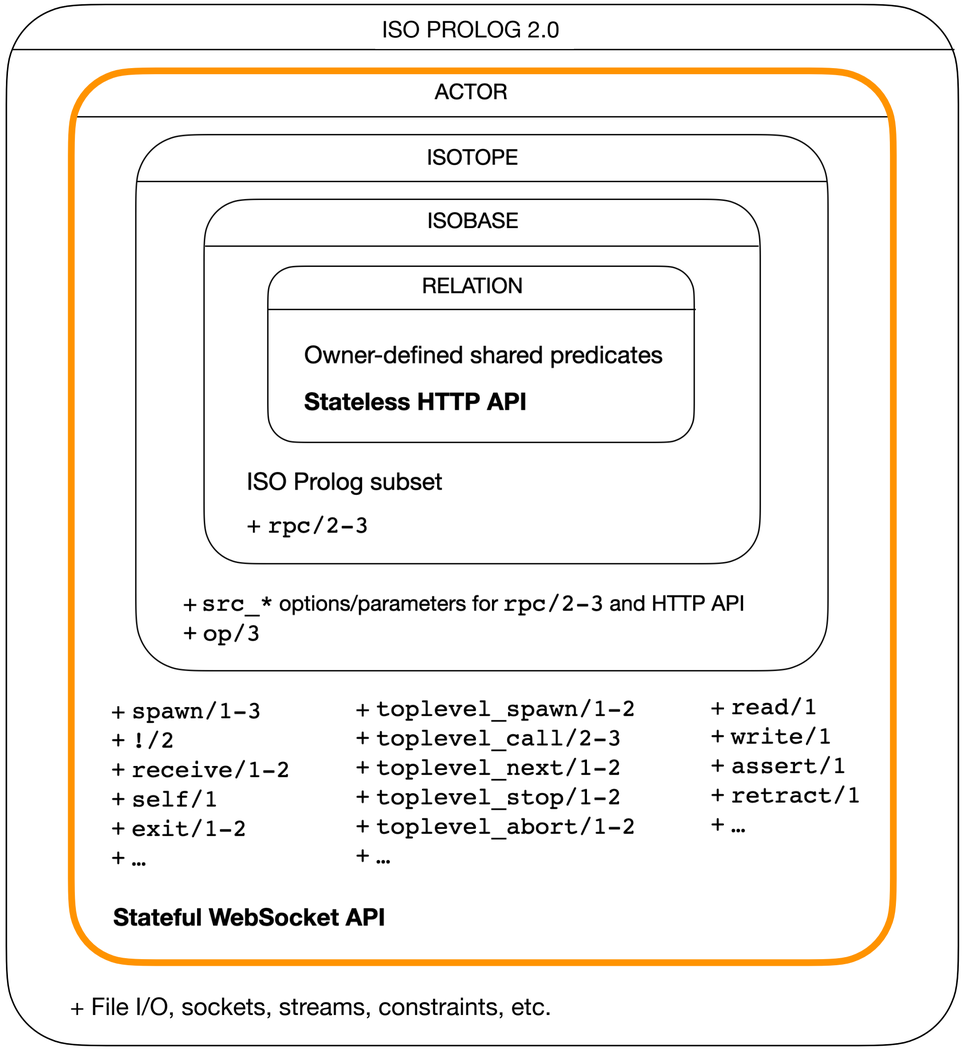
Who might do it?

Who might lead the effort?


Semantic Web - W3C
https://www.w3.org/standards/semanticweb/
Leading the web to its full potential
STANDARDS
PARTICIPATE
MEMBERSHIP
ABOUT W3C
Semantic Web
Skip
SEMANTIC WEB
On this page →
technology topics
news
upcoming events and talks
In addition to the classic “Web of documents” W3C is helping to build a technology stack to support a “Web of data,” the sort of data you find in databases. The ultimate goal of the Web of data is to enable computers to do more useful work and to develop systems that can support trusted interactions over the network. The term “Semantic Web” refers to W3C’s vision of the Web of linked data. Semantic Web technologies enable people to create data stores on the Web, build vocabularies, and write rules for handling data. Linked data are empowered by technologies such as RDF, SPARQL, OWL, and SKOS.
Linked Data
The Semantic Web is a Web of data — of dates and titles and part numbers and chemical properties and any other data one might conceive of. RDF provides the foundation for publishing and linking your data. Various technologies allow you to embed data in documents (RDFa, GRDDL) or expose what you have in SQL databases, or make it available as RDF files.
Vocabularies
At times it may be important or valuable to organize data. Using OWL (to build vocabularies, or “ontologies”) and SKOS (for designing knowledge organization systems) it is possible to enrich data with additional meaning, which allows more people (and more machines) to do more with the data.
Query
Query languages go hand-in-hand with databases. If the Semantic Web is viewed as a global database, then it is easy to understand why one would need a query language for that data. SPARQL is the query language for the Semantic Web.
Inference
Near the top of the Semantic Web stack one finds inference — reasoning over data through rules. W3C work on rules, primarily through RIF and OWL, is focused on translating between rule languages and exchanging rules among different systems.
Vertical Applications
W3C is working with different industries — for example in Health Care and Life Sciences, eGovernment, and Energy — to improve collaboration, research and development, and innovation adoption through Semantic Web technology. For instance, by aiding decision-making in clinical research, Semantic Web technologies will bridge many forms of biological and medical information across institutions.
Web Programming
In fruitful collaboration with the ISO Prolog working group, W3C has created a web logic programming language called Web Prolog. Strictly based on Prolog, but extended with primitives for concurrency and distribution inspired by Erlang, Web Prolog offers a very capable tool for the development of web-based applications of symbolic AI, creating something of a bridge between logic and control.

Stake holders?

Web Prolog as profile of a *future* general-purpose ISO Prolog 2.0?



Paulo Moura

The last thing we need is for Prolog to be downplayed as a Web thing. Prolog is a general programming language and that's the vision a lot of people in the community is working hard to establish. Trying to sell Prolog as a Web silver bullet would only backfire in the same way Prolog association with the hype surrounding AI backfired in the past. While the Web can be a stage to increase logic programming marketshare and mindshare, is not and should not be, in my view, the end game.

Jan Wielemaker

Prolog is quite suitable for the web, but in my experience Prolog is used for a very diverse set of tasks, many of which need Prolog more than the web. I'd love to see Web Prolog fly, in which case it is not unlikely to become more or less detached from the Prolog systems we see targeting other applications.

End of main talking points...

Add-ons

A. Implementing the stateless HTTP API and rpc/2-3

B. Avoiding spurious recomputation in the HTTP API

C. Implementing actors on top the Threads (draft) standard

D. Implementing first-class toplevels on top of actors

Other resources: http://torbjornlager.github.io

Implementing the stateless HTTP API and rpc/2-3

Based on a web server

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

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

node_controller_http(Request) :-
    http_parameters(Request, [
        goal(GoalAtom, [atom]),
        template(TemplateAtom, [default(GoalAtom)]),
        offset(Offset, [integer, default(0)]),
        limit(Limit, [integer, default(10000000000)]),
        format(Format, [atom, default(json)])
    ]),
    atomic_list_concat([GoalAtom,+,TemplateAtom], GTAtom),
    read_term_from_atom(GTAtom, Goal+Template, []),
    compute_answer(Goal, Template, Offset, Limit, Answer),
    respond_with_answer(Format, Answer).

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

Using of findnsols/4 and offset/2 to compute answers

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

```
?- offset(9, between(1, 12, I)).
I = 10 ;
I = 11 ;
I = 12.
?-
```

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

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

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

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

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

There you have your stateless HTTP API!

How is NDRPC implemented?

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

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

rpc(URI, Goal, Options) :-
    parse_url(URI, Parts),
    term_variables(Goal, Vars),
    Template =.. [v|Vars],
    format(atom(GoalAtom), "(~p)", [Goal]),
    format(atom(TemplateAtom), "(~p)", [Template]),
    option(limit(Limit), Options, 10000000000),
    rpc_7(Template, 0, Limit, GoalAtom, TemplateAtom, Parts, Options).

rpc_7(Template, Offset, Limit, GoalAtom, TemplateAtom, Parts, Options) :-
    parse_url(ExpandedURI, [
        path('/call'),
        search([goal=GoalAtom, template=TemplateAtom,
                offset=Offset, limit=Limit, format=prolog])
      | Parts
    ]),
    setup_call_cleanup(
        http_open(ExpandedURI, Stream, Options),
        read(Stream, Answer),
        close(Stream)),
    rpc_8(Answer, Template, Offset, Limit, GoalAtom, TemplateAtom, Parts, Options).

rpc_8(success(Slice, true), Template, Offset, Limit, GoalAtom, TemplateAtom, Parts, Options) :-  !,
    (   member(Template, Slice)
    ;   NewOffset is Offset + Limit,
        rpc_7(Template, NewOffset, Limit, GoalAtom, TemplateAtom, Parts, Options)
    ).
rpc_8(success(Slice, false), Template, _, _, _, _, _, _) :-
    member(Template, Slice).
rpc_8(failure, _, _, _, _, _, _, _) :- fail.
rpc_8(error(Error), _, _, _, _, _, _, _) :- throw(Error).
```

A performance problem with spurious recomputations

```
?- 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.
?-

?- time(rpc('http://n5.org', (sleep(1), X=foo ; X=bar))).
% 1,984 inferences, 0.001 CPU in 1.006 seconds
X = foo ;
% 18 inferences, 0.000 CPU in 0.000 seconds
X = bar.
?-

?- time(rpc('http://n5.org', (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.
?-

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

Fixable!

See Add-on B

Menu

Avoiding spurious recomputation in the HTTP API

Avoiding spurious recomputation in the HTTP API

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

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

node_controller_http(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)]).
```

Avoiding spurious recomputation in the HTTP API

```
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)))
    ]).
```

```
:- 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.


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

Implementing actors on top the Threads (draft) standard

Implementing the ACTOR profile...

... on top of the ISO Prolog Threads (draft) standard

To implement these:

- spawn(+Goal, -Pid, +Options)
- self(?Pid)
- +Pid ! +Message
- receive(+ReceiveClauses, +Options)
- exit(+Pid, +Reason)

... you would need these:

- thread_create(+Goal, -ID, +Options)
- thread_self(?ID)
- thread_send_message(+ID, +Message)
- thread_get_message(+ID, ?Message, +Options)
- thread_signal(+ID, +Goal)
- thread_detach(+ID)
- thread_property(?ID, ?Property)

- thread_local/1 (directive)

... but you won't need these:

- message_queue_create(-Queue)
- message_queue_create(-Queue, +Options)
- message_queue_destroy(+Queue)

- mutex_create(?Mutex)
- mutex_create(-Mutex, ++Options)
- mutex_destroy(+Mutex)

- thread_cancel(+Thread)
- thread_default(?Option)
- thread_exit(+Term)
- thread_join(+Thread, -Status)
- thread_peek_message(-Message)
- thread_peek_message(+Queue, -Message)
- thread_set_default(++Option)
- thread_sleep(+Seconds)

- is_thread(?Term)
- with_mutex(+Mutex, +Goal)

Maybe you shouldn't do it in this way?

Implementing spawn/2-3

```
:- 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)).
```

```
:- 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)).
```

Implementing spawn/2-3

```
:- 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)).
```

```
:- 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)).
```

Implementing exit/1 and exit/2

```
%!  exit(+Reason)
%
%   Exit the current process with a reason.

:- dynamic exit_reason/2.

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

```
%!  exit(+Pid, Reason) is det.
%
%   Exit the actor known as Pid with a reason.

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

Implementing exit/1 and exit/2

```
%!  exit(+Reason)
%
%   Exit the current process with a reason.

:- dynamic exit_reason/2.

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

```
%!  exit(+Pid, Reason) is det.
%
%   Exit the actor known as Pid with a reason.

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

Implementing !/2 and send/2

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

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

Implementing !/2 and send/2

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

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

Implementing receive/1-2

```
:- op(1000, xfy, if).

:- 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({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
    ).
```

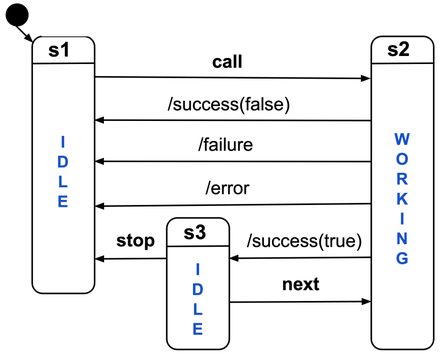
Implementing receive/1-2

```
:- op(1000, xfy, if).

:- 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(Goal), Options, true),
        call(Goal)
    ).
```

```
select_body({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
    ).
```

Implementing first-class toplevels on top of actors

Implementing the toplevel_* predicates (1/2)



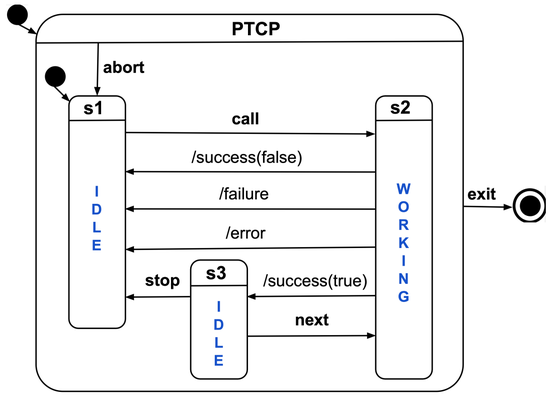

```
toplevel_call(Pid, Goal) :-
    toplevel_call(Pid, Goal, []).

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

toplevel_next(Pid) :-
    Pid ! '$next'.

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

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

toplevel_spawn(Pid, Options) :-
    self(Self),
    spawn(state_1(Pid, Self), Pid, Options).

state_1(Pid, Parent) :-
    receive({
        '$call'(Goal, Options) ->
            option(template(Template), Options, Goal),
            option(offset(Offset), Options, 0),
            option(limit(Limit), Options, 10 000 000 000),
            state_2(Goal, Template, Offset, Limit, Pid, Answer),
            Parent ! Answer,
            (   arg(3, Answer, true)
            ->  state_3
            ;   true
            )
    }),
    state_1(Pid, Parent).
```

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

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


state_3 :-
    receive({
        '$next' ->
            fail ;
        '$stop' ->
            true
    }).
```

More is needed!

Implementing the toplevel_* predicates (2/2)




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

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

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

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

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

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

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

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

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

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)
    ).
```

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

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


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

Menu