Internet and WWW Programming using Computational Logic Systems (PART I)

Manuel Hermenegildo
(joint work with Daniel Cabeza)
{herme,dcabeza}@dia.fi.upm.es

http://www.clip.dia.fi.upm.es/

CLIP Group
School of Computer Science
Technical University of Madrid (UPM)

Workshop on Logic Programming and the Internet
December 19–20, 1996
Imperial College, London, U.K.

Work supported in part by ESPRIT projects “ACCLAIM” and “PARFORCE” and by CICYT/MEC grant “ELLA.” Sponsored by Compulog-Net III.

LP/CLP, the Internet, and the WWW

- Can Logic and Constraint Logic Programming be an attractive alternative for Internet/WWW programming?
- Shared with other net programming tools:
  - dynamic memory management,
  - well-behaved structure and pointer manipulation,
  - robustness, compilation to architecture-independent bytecode, ...
- In addition:
  - powerful symbolic processing capabilities,
  - dynamic databases,
  - search facilities,
  - grammars,
  - sophisticated meta-programming / higher order,
  - easy code (agent) motion,
  - well understood semantics, ...
- Most LP/CLP systems already have Internet connection capabilities (e.g., socket interfaces).
- Some "glue code" needed: we present several techniques for "filling in the gaps" (available as public domain libraries).
- We also try to answer the question: is there anything fundamental missing in current LP/CLP systems?

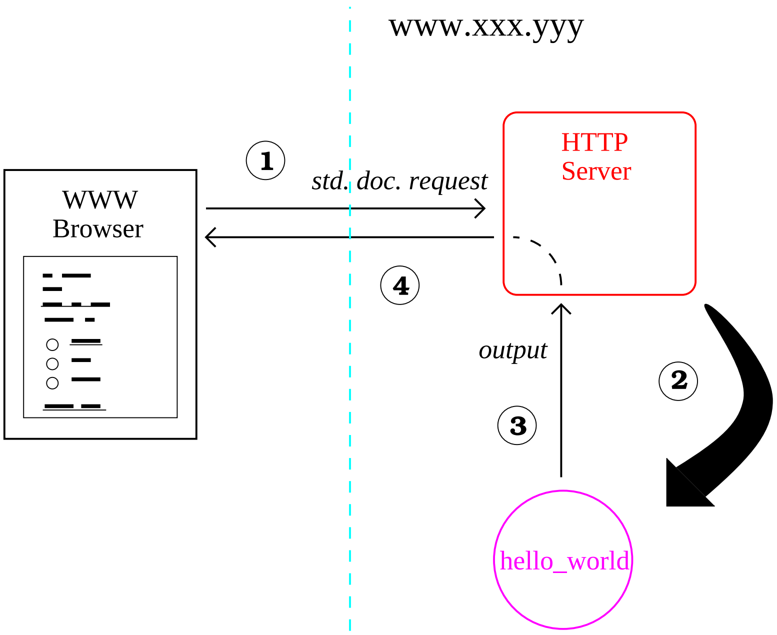
Global Outline

- PART I: *WWW programming*
  - Writing cgi-scripts.
  - Seeing HTML structured documents as Herbrand terms.
  - Producing HTML *forms*.
  - Writing form handlers.
  - Accessing and parsing WWW documents.
  - Accessing code posted at HTTP addresses.
- PART II: *Distributed (and agent) programming*
  - Blackboards.
  - A design for a distributed LP/CLP language.
  - Modeling and accessing information servers (active modules).
  - Implementing distributed variable-based communication using attributed variables.

Systems Discussed

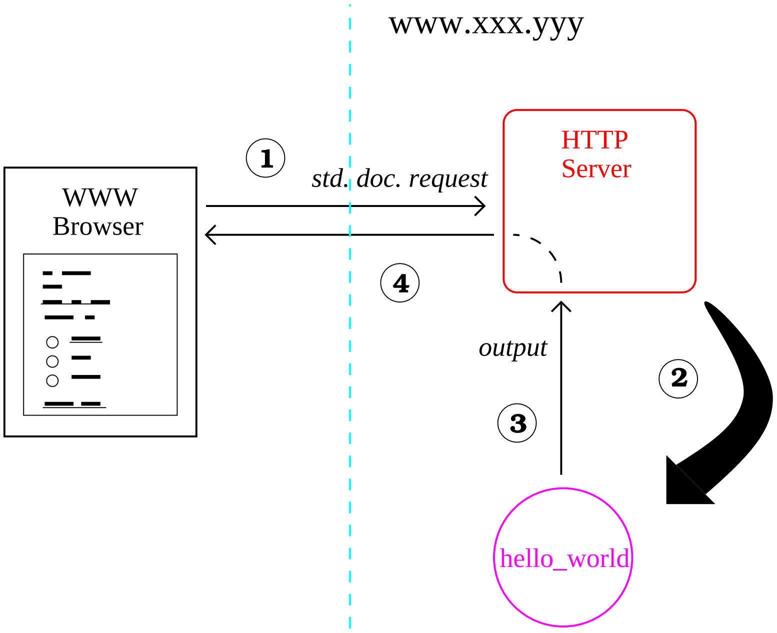
- By "LP/CLP" systems we mean: Quintus / LPA / SICStus / IF-Prolog / Eclipse / BinProlog / PrologIV / CHIP / ...
- We also refer to CIAO – the UPM Concurrent CLP system:
  - ◇ Built as an extension of ISO-Prolog.
  - ◇ Platform for designing
    - * useful extensions of LP and CLP towards concurrency, higher-order, distribution, ...
    - * advanced compilation and debugging technology.
  - ◇ Libraries also built to provide (as much as possible of) this functionality on standard LP/CLP systems.
  - ◇ E.g., answers the question of what really needs to be added to/changed in current systems.
- CIAO Libraries (freely available, and in different stages of development) include:
  - ◇ PiLLoW: WWW/HTML interface.
  - ◇ prolog˙shell: Prolog shell scripts
  - ◇ Distribution (blackboards, concurrency, agents, ...)
  - ◇ PLAI: Global analysis (including type checking/inferencing)
  - ◇ APC: Global optimization (including parallelization)

Writing Basic CGI-bin Applications



1. The address of the CGI application is selected in a browser. This address is a standard URL, e.g.
   `http://www.xxx.yyy/cgi_bin/hello_world`.
   The browser sends it to the corresponding HTTP server.
2. The executable "`hello_world`" (in directory `cgi_bin`) is started by the HTTP server.
3. The output (stdout) of the executable (which has to be in HTML –or MIME– format) is taken by the HTTP server,
4. and passed on to the Browser, which displays it.

Writing Basic CGI-bin Applications in LP/CLP



- The most basic code can be something like:

```
main(_) :-
        write('Content-type: text/html'), nl, nl,
        write('<HTML>'),
        write('Hello <B>world</B>.'),
        write('</HTML>'), nl.
```

- And the executable generated, e.g., by:

```
?:- save('hello_world'), main(_), halt.
```

Simplifying Things: LP/CLP scripts

- “Scripting” languages (perl, csh, ...) popular for writing CGI apps:
  - Often only small- to medium-sized apps.
  - The network is slow anyway.
  - No compilation necessary.
  - Small “executable” size (source file!).
- LP/CLP systems are excellent candidates for scripting languages: built-in grammars, databases, ...
- Most LP/CLP systems can easily be used as scripting languages: some have it (e.g., CIAO), others can be adapted easily with some user-level code (e.g., we have adapted the CIAO solution for SICStus and made it freely available on ftp):

```
#!/usr/local/bin/lpshell
main(_) :-
        write('Content-type: text/html'), nl, nl,
        write('<HTML>'),
        write('Hello <B>world</B>.'),
        write('</HTML>'), nl.
```

- `lpshell` is an executable (e.g., a saved state) which skips the first line, loads (consults or compiles) the rest of the file, and starts at `main/1`.
- Can easily be made to “cache” compilations using bytecode files.
- (very useful also to write “filters” –e.g., for unix pipes–, etc.)

Relating HTML code and Prolog Terms

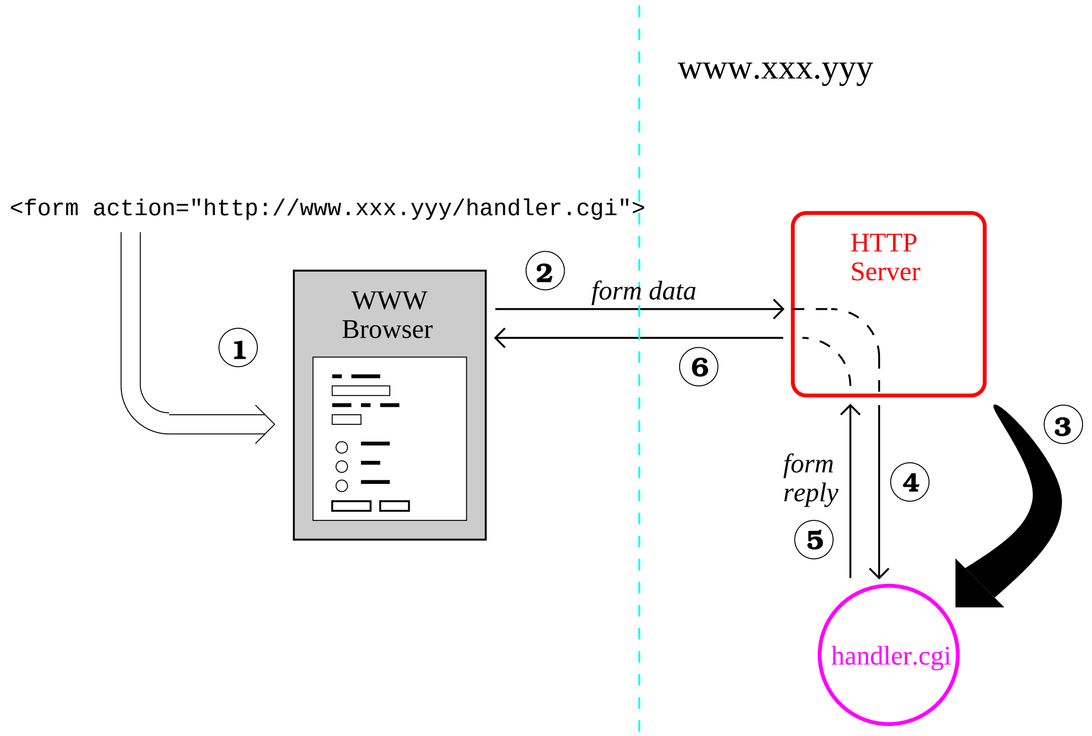
- HTML is structured: it is possible to reflect this structure as Prolog terms.
- Allows viewing any WWW page as a Herbrand term and manipulating it easily.
- Ideally, provide bidirectional conversion between a string representing the HTML code and its term representation (e.g., with DCGs).
- E.g, predicates for this purpose provided in PiLLoW:
  - `output_html(`*F*`)` – sends to the standard output the text corresponding to the HTML term *F*.
  - `html2terms(`*ASCII*`,` *Terms*`)` – Relates a list of HTML terms and a list of ASCII characters (reversible).
- Example:

```
#!/usr/local/bin/lpshell
:- use_module('/usr/local/src/pillow/html.pl').
main(_) :-
    T = [ cgi_reply,
          html( [ 'Hello',
                  b(world)
               ] )
        ]),
    output_html(T).
```

Relating HTML code and Prolog Terms (Contd.)

- P*i*LL*o*W general HTML structures (can represent any HTML code):
  - ◇ *Name* `$` *Atts* (‘`$/2`’ is defined as an infix, binary operator.)
    `img$[src='images/map.gif',alt='A map',ismap]` ⇒
    `<img src="images/map.gif" alt="A map" ismap>`
  - ◇ *name*(*Text*) (Term with functor *name*/1)
    `address('clip@dia.fi.upm.es')` ⇒
    `<address>clip@dia.fi.upm.es</address>`
  - ◇ *name*(*Atts*, *Text*) (Term with functor *name*/2)
    `a([href='http://www.xx.y/'],'XX home')` ⇒
    `<a href="http://www.xx.y/">XX home</a>`
  - ◇ `env`(*Name*, *Atts*, *Text*)
    `env(a,[href='http://www.xx.y/'],'XX home')` ⇒
    `<a href="http://www.xx.y/">XX home</a>`
- Also, specific structures to simplify HTML creation.

Responding to Input: Forms



- A form is a standard HTML document (supported by all browsers) with
  - "fields" which can be filled in (each has a *name*),
  - a "submit" button,
  - URL of CGI app. that will handle the input (the "handler")
- Operation (when hitting the "submit" button):
  1. We assume the URL of the handler is `http://www.xxx.yyy/handler.cgi`.
  2. Handler URL and form data (input) are passed to server.
  3. Server starts handler and passes form data (via stdin or standard file name) associating field names to entered values.
  4. The handler produces the appropriate reply,
  5. which is passed back to the browser.

Writing Form Handlers in LP/CLP

- Use same techniques as with standard CGI apps.
- Only complication is form data parsing (names/values).
- Good solution: implement a parser (easy in LP/CLP) and produce an attribute-value pair list or dictionary.
- Enables the symbolic treatment of form data, hiding the low-level protocol behind.

E.g., predicates provided in P*i*LL*o*W:

- **`get_form_input(`*Dic*`)`** Translates input from the form to a dictionary *Dic* of *attribute*=*value* pairs. This is implemented using a simple DCG parser.
  `get_form_input(Dict)` ⇒ `Dict = [name='Anna', age=23]`
- **`get_form_value(`*Dic*, *Var*, *Val*`)`** Gets value *Val* for attribute *Var* in dictionary *Dic*.
- **`form_empty_value(`*V*`)`** Useful to check that a value *V* from a text area is empty.
- **`my_url(`*URL*`)`** Returns in *URL* the Uniform Resource Locator (WWW address) of this form.

A browser can be used as a graphical interface!

Writing Form Handlers in LP/CLP: Example

- A simple form:

```
<html>
<hr>
<h2>Please enter input (person_name):</h2>
<form method="POST"
 action="http://www.xxx.yyy/simple_handler.cgi">
<input type="text" name="person_name" size="40">
<input type="submit" value="Submit">
</form>
<hr>
</html>
```

- A simple form handler (`simple_handler.cgi`):

```
#!/usr/local/bin/lpshell
:- use_module('/usr/local/src/pillow/html.pl').
main(_) :-
    get_form_input(Input),
    get_form_value(Input,person_name,Name),
    output_html([
        cgi_reply,
        start,
        --,
        heading(2,'You submitted the name: '),
        em(Name),
        --,
        end]).
```

Producing Forms from Programs: Example

- The form itself can be the result of running a program:

```
#/usr/local/bin/lpshell
:- use_module('/usr/local/src/pillow/html.pl').
main(_) :-
  output_html([
    cgi_reply,
    start,
    --,
    heading(2,'Please enter input (person_name):'),
    start_form('http://www.xx/simple_handler.cgi'),
    input(text,[name=person_name,size=40]),
    input(submit,[value='Submit']),
    end_form,
    --,
    end]).
```

Combining the Form Producer and Handler

```
#!/usr/local/bin/lpshell
:- use_module('/usr/local/src/pillow/html.pl').
main(_) :-
  get_form_input(Input),
  get_form_value(Input,person_name,Name),
  output_html([
      cgi_reply,
      start,
      --,
      heading(2,'You submitted the name: '),
      em(Name),
      --,
      heading(2,'Please enter input (person_name):'),
      start_form,
      input(text,[name=person_name,size=40]),
      input(submit,[value='Submit']),
      end_form,
      --,
      end]).
```

A Phones Database

```
main(_) :-
        get_form_input(Input),
        get_form_value(Input,person_name,Name),
        response(Name,Response),
        output_html([
          cgi_reply,
          start,
          title('Simple CLIP telephone database'),
          center( [
            image('http://localhost/images/clip.gif'),
            heading(2,'Simple CLIP telephone database'),
            --,
            Response,
            start_form,
            'Click here, enter name of clip member,
            and press Return:',
            \\,
            input(text,[name=person_name,size=20]),
            --,
            end_form,
            image('http://localhost/images/pillow_d.gif') ]),
          end ]).

response(Name, []) :- form_empty_value(Name), !.
response(Name,
         ['Phone number for ', bf(Name), ' is ', Info, --]) :-
         phone(Name,Info), !.
response(Name,
         ['No phone number available for ', bf(Name), '.', --]).

%% Database
phone('CLIP',   '336-7448').
phone('Paco',   '554-5225').
phone('Daniel', '460-0569').
```

Accessing WWW/Internet documents from LP/CLP

- The HTTP, FTP, etc. protocols are ASCII protocols which can be added relatively easily to any system (provided it has a socket interface or equivalent).
- Applications: search tools, content analyzers, reading html templates, etc. Also access to remote modules via WWW:
  `:- use_module('http://www.xx.y/prolog/p.pl')`
- E.g., P*i*LL*o*W protocol support:
  - ◇ `fetch_url((URL),` *Request*`,`*Response*`)` –
  - ◇ Some (Request) options:
    - * `head`: only interested in the header.
    - * `timeout(`*Time*`)`: specifies number of seconds for timeout (fails).
    - * `if_modified_since(`*Date*`)`
    - * `authorization(`*Scheme*`,`*Param*`)`
  - ◇ Some possible elements of (Response):
    - * `content((Content))`: document as a list of characters.
    - * `status((Type),(Code),(Phrase))`.
    - * `last_modified((Date))`.
    - * `expires((Date))`.
    - * `location((URL))` (document has moved).
  - ◇ `url_info((URL),` *Info*`)`: parses a URL.
- Example:

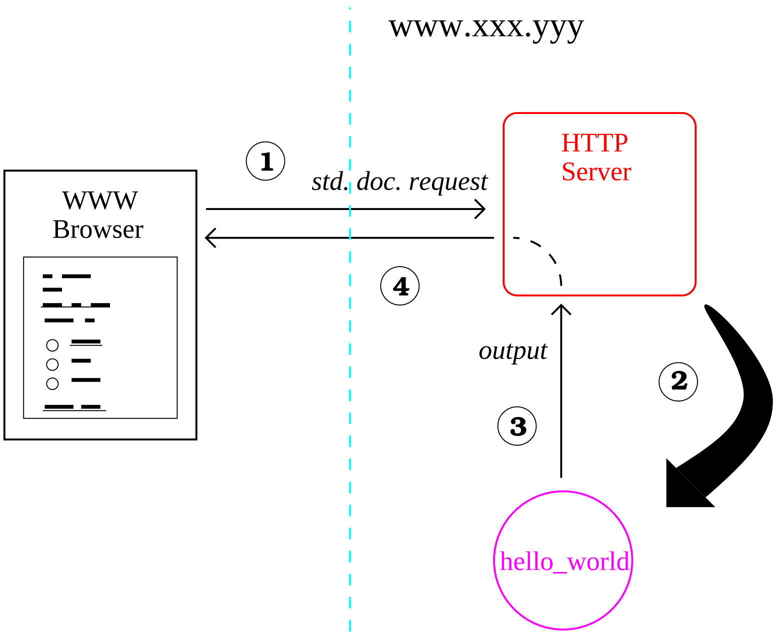
```
?:- url_info('http://www.xx/foo.html',UI),
fetch_url(UI,[],R),
member(content(C),R), html2terms(R,Terms).
```

Accessing WWW/Internet: Example

```
check_links(URL,BadLinks) :-
        url_info(URL,URLInfo),
        fetch_url(URLInfo,[],Response),
        member(content_type(text,html,_),Response),
        member(content(Content),Response),
        html2terms(Content,Terms),
        check_source_links(Terms,URLInfo,[],BadLinks).

check_source_links([],_,BL,BL).
check_source_links([E|Es],BaseURL,BL0,BL) :-
        check_source_links1(E,BaseURL,BL0,BL1),
        check_source_links(Es,BaseURL,BL1,BL).
check_source_links1(env(a,AnchorAtts,_),BaseURL,BL0,BL) :-
        member((href=URL),AnchorAtts), !,
        check_link(URL,BaseURL,BL0,BL).
check_source_links1(env(_Name,_Atts,Env_html),BaseURL,BL0,BL) :- !,
        check_source_links(Env_html,BaseURL,BL0,BL).
check_source_links1(_,_,BL,BL).

check_link(URL,BaseURL,BL0,BL) :-
        url_info_relative(URL,BaseURL,URLInfo), !,
        fetch_url_status(URLInfo,Status,Phrase),
        ( Status \== success ->
          name(P,Phrase),
          name(U,URL),
          BL = [badlink(U,P)|BL0]
        ; BL = BL0
        ).
check_link(_,_,BL,BL).

fetch_url_status(URL,Status,Phrase) :-
        fetch_url(URL,[head,timeout(20)],Response), !,
        member(status(Status,_,Phrase),Response).
fetch_url_status(_,timeout,timeout).
```

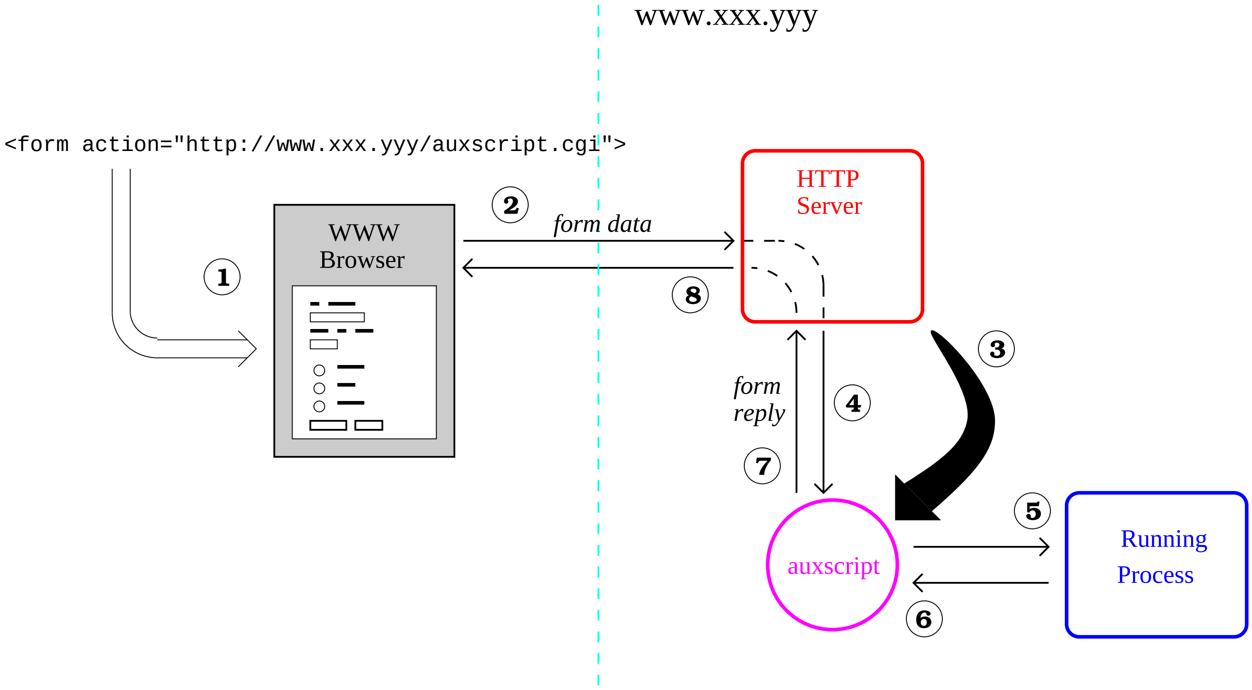
Limitations of CGI

- The cgi-bin interface dictates that a handler of a form starts and terminates for each interaction.
- Thus, form handlers in principle do not have state.
- State can in fact be passed through the form interface (using info in hidden fields).
- However, for a large application, staring and stopping can be very inefficient.



Solving the Limitations of CGI

- Standard solution: make the application be a permanently running process. A small CGI script (generally written in perl) connects and disconnects form it for every interaction.



- In LP/CLP:
  - ◇ The running process is a standard application.
  - ◇ The CGI executable can be an LP/CLP script.
  - ◇ Communication by standard means: sockets, blackboards, etc.
  - ◇ Several solutions proposed for dealing with several running sessions at the same time (in essence, concurrency is needed).
- "Active modules" can be used well for this purpose.

Java-related Functionality

- Automatic code downloading:
  - Can be easily done using for a particular browser (e.g., as a netscape "plug-in", or using Mosaic's API, as "LogicWeb").
  - Can actually be done independently of the browser.
- Complex user interfaces:
  - Can be done using a tcl/tk "plug-in".
  - More standard: generate Java code from Prolog!
- Also, use a Prolog to Java compiler (but execution probably slow).

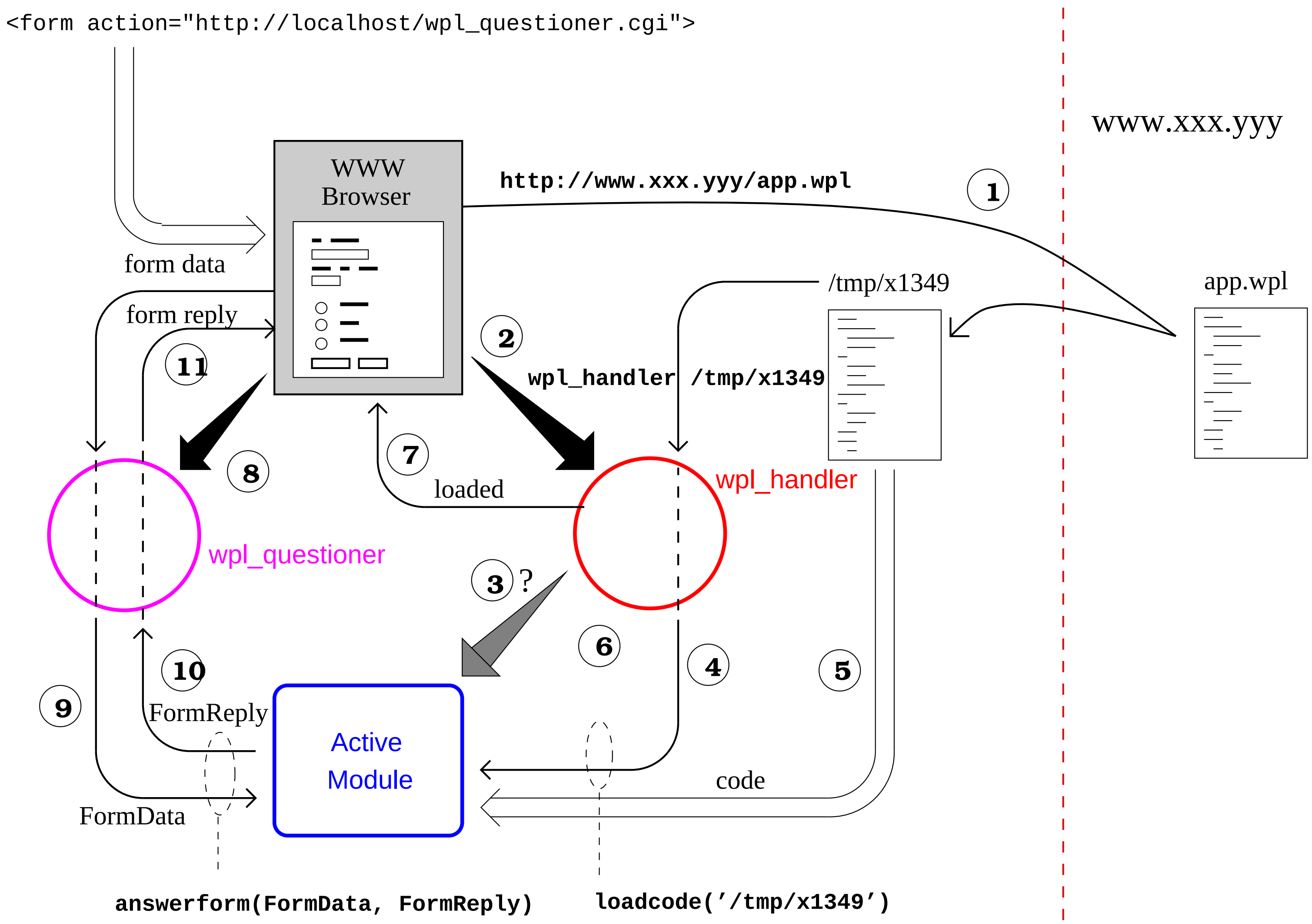
Automatic Code Downloading for Local Execution

- Using only the facilities presented, automatic LP/CLP code downloading for local execution is possible, using generic browsers.
- By simply clicking on a WWW pointer, and transparently for the user, remote code is automatically downloaded and locally queried via forms.
- Prerequisites:
  - The HTTP server on the server machine is configured to give a `mime.type` of `application/x-prolog` to the files with WWW-downloadable LP code.
  - The browser is configured to start helper `wpl_handler` when receiving data of type `application/x-prolog` (this application starts the LP engine as an active module).
  - There is a local cgi-bin executable `wpl_questioner.cgi` (which uses that active module).

Automatic Code Downloading Procedure

1. A click in a link of the query form starts the downloading of the code (alternatively using `multipart/mixed` mime type).
2. The broser starts a `wpl_handler` as the document has type `application/x-prolog`.
3. The `wpl_handler` process starts a Prolog engine (configured as an active module) if necessary.
4. Then, through a call "`loadcode(`*File*`)`" the handler asks the active module to read the code.
5. The active module reads the code and compiles it.
6. `wpl_handler` waits for the active module to complete the compilation and writes a "done" message to the browser.
7. The browser receives the "done" message.
8. Now, when the "submit" button in the form is pressed, the browser starts a `wpl_questioner` as a handler of the form data.
9. The `wpl_questioner` process translates form data to a dictionary *FormData*, passing it to the active module through a call `answerform(`*FormData*`,`*FormReply*`)`.
10. The active module processes this request, and returns in *FormReply* a WWW page (as a term) containing the answer.
11. The `wpl_questioner` process translates *FormReply* to raw HTML and gives it back to the browser, dying afterwards. Subsequent queries proceed at 8.

Automatic Code Downloading Procedure – Figure


<form action="http://localhost/wpl_questioner.cgi">
www.xxx.yyy
WWW
Browser
http://www.xxx.yyy/app.wpl
1
form data
form reply
/tmp/x1349
app.wpl
11
2
wpl_handler /tmp/x1349
8
7
loaded
wpl_handler
wpl_questioner
3 ?
6
4
5
10
9
FormReply
Active
Module
code
FormData
answerform(FormData, FormReply)
loadcode('/tmp/x1349')

Some Related Work (LP/CLP + WWW)

- PiLLoW includes previous work in `html.pl`, F. Bueno's WWW Chat version, and L. Naish's NU-Prolog forms.
- Also, K. Bowen's port of `html.pl` to ALS Prolog, which provides group processing of forms and an alternative to our use of active modules.
- LogicWeb (Loke and Davidson): extends LP to model the concept of WWW pages, incorporating in them programmable behaviour and state. Powerful module management. Higher level than PiLLoW.
- Szeredi's multiple request handling through or-parallelism.
- ECLiPSe HTTP support library (by replacing HTTP server).
- Many LP/CLP Internet applications shown in the JICSLP workshop.

More recently:

- Prolog to Java compilers (Bart Demoen's, Minerva, etc.)
- WebPlug: proposes simple solution (not needing sockets) to persistent CGI processes.
- Other higher-level interfaces: ProWeb (persistence, templates), generation of interfaces from database schemata [A.Porto],...

Some Conclusions

Conclusions/Other Issues:

- LP/CLP concepts/technology well suited for Internet applications, with some support. Many applications already developed *WebChat*, Rent Advisor and others in the JICSLP Workshop, ...
- The PiLLoW library has been designed to provide basic help in this. It is available from:
  `http://www.clip.dia.fi.upm/miscdocs/pillow/pillow.html`
- Underlying support for **concurrency** and **distribution** (e.g., &-Prolog/CIAO, BinProlog/μ^2-Prolog, ...) has many advantages: overlapping requests, distributed agent programming, etc.

CIAO WebChat

CIAO WEBCHAT Interface

what is the capital of spain ?

`madrid.`

Done.

Please enter question (end question with ?):

Submit (text and/or options) | Reset (text and options)

Question processing options:

- Normal output (answers)
- Report syntactic and semantic analysis
- No output
- Report actual answer computation time (walltime)

Execution mode options:

- Use local execution
- Use distributed execution for performance
- Use distributed execution for remote DB access
- Report execution option processing time (walltime)

Under automatic construction guided by [Manuel Hermenegildo]