A Logic Programming Infrastructure for Remote Execution, Mobile Code and Agents

Paul Tarau
Université de Moncton
tarau@info.umoncton.ca

Veronica Dahl
Simon Fraser University
veronica@cs.sfu.ca

Koen De Bosschere
Universiteit Gent
kdb@elis.rug.ac.be

Abstract

We describe a set of programming patterns used for implementing a scalable infrastructure which supports remote execution mechanisms, mobile code and agents in a distributed logic programming framework. The particular focus of this paper is on the use of BinProlog's strong metaprogramming abilities. Some advanced logic programming constructs as intuitionistic implication, high-order call/N cooperate with encapsulated socket-level constructs for maximum configurability and efficiency. We show that strong metaprogramming is not a security threat if used through a set of filtering interactors *which allow source level implementation of arbitrary security policies. Mobile code is implemented in a scalable way through a set of distributed client+server pairs interconnected through a master server acting only as an address exchange broker for peer-to-peer interactors. We have thoroughly tested our programming patterns and design principles through a realistic implementation in a widely used, freely available Prolog system (*`http://clement.info.umoncton.ca/BinProlog`*) as well as with its Java peers built on top of our unification enhanced Java based Linda implementation (*`http://clement.info.umoncton.ca/LindaInteractor`*).*

Keywords: mobile code, remote execution, metaprogramming, agents, Linda coordination, blackboard-based logic programming, Prolog, distributed programming, virtual worlds, scalable and secure Internet programming, Prolog/Java embedding

1 Introduction

Although the Internet has been designed to survive nuclear war and its underlying packet switching technology is intrinsically peer-to-peer, fault-tolerant and scalable, successive higher level networking and programming layers have given up (often too easily) these abilities. Among the most annoying *and* at the same time the most pragmatically well thought decisions dominating the world of the Internet, at various programming and application development layers:

- classical client/server architectures have given up scalability for simplicity of programming and synchronization;
- programming languages/operating systems have given up powerful remote execution mechanisms for reasons ranging from security concerns to legacy single user/single process assumptions.

In this paper, we will show how some standard and some non-standard Logic Programming language tools will be used to elegantly get back the full potential of the Internet for building scalable, peer-to-peer, programmable multi-user communities. The virtual layer we will build is based on a small set of very-high level programming constructs making essential use of meta-programming, which is seen here as the ability to view information either as code or as data, and efficiently switch between these views, on demand. In particular, we will show how powerful remote execution mechanisms, agents and mobile code can be expressed in this framework.

With important resources at their disposal, operating system/computer/network companies have found their, often complex and costly ways to get back some of the original power, hidden under various layers of software/hardware. Despite the existence of practical solutions to some of these problems, we hope that the unifying approach we propose in this paper will help clarify the logical structure of future peer-to-peer, scalable virtual communities, where the distinction between human and non-human agents becomes less and less obvious. We are also convinced that the power of the knowledge/processing component of logic programming will help the efforts towards intelligent network/mobile agent programming more effective.

We will describe our constructs as built on the top of the popular Linda [3] coordination framework, enhanced

0-8186-7967-0/97 $10.00

with unification improved pattern matching [4], although our actual implementation is based on a more general *co-ordination logic*, allowing negotiation of removal of tuples as well as of suspension between user intents at *definition time*, when an object is stored on the (shared and synchronized) *blackboard* as well as at *reference time* when it is retrieved associatively[10].

```
out(X):     puts X on the server
in(X):      waits until it can take an object
            matching X from the server
all(X,Xs):  reads the list Xs matching X
            currently on the server
run(Goal):  starts a (remote) thread
            executing Goal
```

The presence of the `all/2` collector compensates for the lack of non-deterministic operations. Note that the only blocking operation is `in/1`. Notice that blocking `rd/1` is easily emulated in terms of `in/1` and `out/1`, while non-blocking `rd/1` is emulated with `all/2`.

2 Simple Agents

A *servant* is one of the simplest possible *agents*:

```
servant(Who):-in(todo(Task)),
  secure(Task),run(Task),
  servant(Who).
```

Note that *servant* is started as a background thread with `run(servant(Name))`. No 'busy wait' is involved, as the servant's thread blocks until `in(todo(Task))` succeeds. In an 'Internet chat' application, for instance, `whisper/2`, defined as

```
whisper(To,Mes):-whoami(From),
  out(todo(mes_to(To,Mes,From))).
```

unblocks the matching

```
  in(todo(mes_to(To,Mes,From)))
```

of a *notifier servant* which then outputs a message by executing `mes_to(To,Mes,From)`.

More generally, distributed event processing is implemented by creating a 'watching' agent for a given pattern.

Note the fundamental link between 'event-processing' and the more general Linda protocol: basically an `out/1` operation can be seen as generating an event and adding it to the event queue while an `in/1` operation can be seen as servicing an event. While usually event-loops switch on numeric or pseudo-numeric event constants in a rather rigid and non-compositional way, Linda-based event dispatching is extensible by adding new patterns. When (naturally) restricted to ground patterns, efficient indexing can be used with performance hits unnoticeable in a network lag context. Linda's `out/blocking in` combination can be seen as automating the complex if-then-else logic of (distributed) hierarchical message dispatching loops through unification.

Remote processing as well as simple security mechanisms are expressed easily, by creating 'command server' thread close to the following code:

```
% remote processing request
please(Who,What):-whoami(ForWho),
  out(please(Who,ForWho,What)).

% a remote command processor
command_server:-whoami(Me),
  repeat,
    in(please(Me,ForWho,What)),
    ( friend_of(Me,ForWho)->
      call(What) % trusted friends
    ; errmes(intruder,ForWho)
    ),fail.
```

Clearly, `command_server` and `notifier` threads can be seen as 'behavior components' of a unique agent. Moreover, they actually might cooperate in a synchronized way if each `please/2` command would trigger a `whisper/2` action to be serviced by a notifier later. In this example security is based on allowing arbitrary execution for trusted (i.e. known to `friend_of/2`) clients of existing server-side code.

Note that by restricting `command_server` to execute only sequences of acceptable known commands, on top of code-as-data sent through `out/1` transactions, arbitrary security mechanisms can be put in place at source level. We will describe this later in more detail.

3 Towards a Web of Worlds

The MOO[1] inspired 'Web of Worlds' metaphor [12, 14] as a set of Linda blackboards storing state information on servers connected over the the Internet allows a simple and secure remote execution mechanism through specialized *server-side* interpreters.

Our implementation is integrated in the freely available BinProlog compiler. A master server on a 'well-known' host/port is used to exchange identification information among peers composed of a client and a 'multiplexing' server .

[1] Multi User Domains (MUDs), Object Oriented - venerable but still well doing ancestors of more recent multi-user Virtual Worlds, which are usually 3D-animation (VRML) based

3.1 Server side code

The code for our 'generic' server, with various components which are overridden through use of intuitionistic implication to obtain customized special purpose servers at user level, follows. Higher order call/N [8], combined to intuitionistic assumptions `'=>>'` are used to pass arbitrary *interactors* to this generic server code.

```
run_server(Port,Fuel):-
  new_server(Port,NewServer),
  register_server(Port),
    server(NewServer)=>>
     server_loop(NewServer,Fuel),
  close_socket(NewServer).

server_loop(Server,Fuel):-
  for(I,1,Fuel),
    interact(Server),
    assumed(server_done(Mes)),!.

interact(Server):-
  default_server_action(Interactor),
  % higher-order call to interactor
  call(Interactor,Server),!.
```

Note the use of a specialized *server-side* interpreter `server_loop`, configurable through the use of higher-order 'question/answer' closures we have called *interactors*.

We have found out that use of intuitionistic implications (pioneered by Miller's work [7]) helps to overcome (to some extent) Prolog's lack of object oriented programming facilities, by allowing us to 'inject' the right interactor into the generic (and therefore reusable) interpreter. BinProlog's `=>>` temporarily assumes a clause in 'asserta' order, i.e. at the beginning of the predicate. The assumption is scoped to be only usable to prove its right side goal and vanishes on backtracking. We refer to [9, 11] for more information on assumptions and their applications.

3.2 Client side code

The following example shows how the same client-side code is used for both for point-to-point and 'broadcast' communication, depending on a `to_all(Pattern)` assumption allowing to select matching targets.

```
ask_server(Question,yes):-
  assumed(to_all(ServerPattern)),!,
  all_servers(ServerPattern,Xs),
  ask_all_servers(Xs,Question).
ask_server(Question,Answer):-
  ask_a_server(Question,Answer).
```

Slightly more complex code (see file extra.pl in the BinProlog distribution) also handles proxy forwarding services transparently.

4 Remote execution mechanisms

Implementation of arbitrary remote execution is easy in a Linda + Prolog system due to Prolog's metaprogramming abilities. No complex serialization or remote procedure/method call packages are needed. In BinProlog (starting with version 5.60) code fetched lazily over the network is cached in a local database and then dynamically recompiled on the fly if usage statistics indicate that it is not volatile and it is heavily used locally.

```
host(Other_machine)=>>
  remote_run(RemoteGoal).

host(Other_machine)=>>
  remote_run(Answer,RemoteGoal).
```

allow remote predicate calls without/with returned answer on the calling site.

```
host(Other_machine)=>>
  rsh(ShellCommand,CharListAnswer).
```

allows using BinProlog to remotely execute shell commands on a remote machine and collect/print the answers at the calling site. Despite Prolog's lack of object oriented features we have implemented code reuse with intuitionistic assumptions [11].

For instance, to iterate over the set of servers forming the receiving end of our 'Web of Worlds', after retrieving the list from a 'master server' which constantly monitors them making sure that the list reflects login/logout information, we simply override `host/1` and `port/1` with intuitionistic implication:

```
ask_all_servers(ListOfServers,Question):-
  member(server_id(_,H,P),ListOfServers),
  host(H)=>>port(P)=>>
    ask_a_server(Question,_),
  fail;true.
```

To specialize a generic server into either a master server or a secure 'chat-only' server which merges messages from BinProlog users world-wide we simply override the filtering step in the generic server's main interpreter loop.

Unrestricted servers: full remote metaprogramming On an Intranet of trusted users and computers, or in different windows of an unconnected PC or workstation a user might want to experiment with a server allowing arbitrary Prolog command execution i.e. full *remote metaprogramming*. One can start a local remote predicate call server with:

```
?-run_unrestricted_server.
```

Note that registered servers are in principle accessible to other users and therefore *not* secure.

We will give next two simple approaches to implement security.

A first approach to security: servers with restricted interactors Here is the code of the chat server:

```
chat_server_interactor(mes(From,Cs),Answer):-
  show_mes(From,Cs,Answer).
chat_server_interactor(ping(T),R):-
  term_server_interactor(ping(T),R).

chat_server:-
  server_interactor(chat_server_interactor)
  =>>run_server.
```

By overriding the default `server_interactor` with our `chat_server_interactor` we tell to `chat_server` that only commands matching known, secure operations (i.e. `mes/2` and `ping/1`) have to be performed on behalf of remote users.

Security is achieved by having specialized meta-interpreters 'filtering' requests on the server side. Intuitionistic assumption is used to override the generic server's *interactor*, i.e. the inner server operation filtering allowed requests, while allowing reuse of most of the generic server's code in an object oriented programming style.

A second approach to security: starting an Intranet specific master server Keeping one's host/port information secret from other users can be achieved by starting a master server local to a secure physical or virtual Intranet. For users behind a firewall, this might actually be the only way to try out these operations as the default master server might be unreachable.

To keep the workload of the master server minimal, only when an error is detected by a client, the master server is asked to refresh its information and possibly remove dead servers from its database.

Interaction with Java Applets. BinProlog starting from version 5.40 communicates with our Java based *Linda Interactors*, special purpose trimmed down pure Prolog engines written in Java which support the same unification based Linda protocol as BinProlog. The natural extension was to allow Java applets to participate to the rest of our 'peer-to-peer' network of BinProlog interactors. As creating a server component within a Java applet is impossible due to Java's (ultra)-conservative security policies we have simply written a receiving-end *servant* close to our example in subsection 2, which pulls out commands from a proxy server on the site where the applet originates from, for seamless integration in our world of peer-to-peer interactors.

5 Mobile Code

We will shortly discuss here the basic Mobile Code facilities we have implemented. Future work will focus on building intelligent agent applications on top of them.

5.1 Remote code fetching

The following operations

```
host(Other_machine)=>>rload(File).
host(Other_machine)=>>code(File)=>>TopGoal.
```

allow fetching remote files `rload/1` or on-demand fetching of a predicate at a time from a remote host during execution of TopGoal. This is basically the same mechanism as the one implemented for Java applet code fetching, except that we have also implemented a caching mechanism, both at predicate level (predicates are cached as dynamic code on the server to efficiently serve multiple clients) and at file level, on the client side.

5.2 Dynamic recompilation

Dynamic recompilation is used on the client side to speed-up heavily used relatively non-volatile predicates. With dynamically recompiled consulted code, listing of sources and dynamic modification to any predicate is available while average performance stays close to statically compiled code (usually within a factor of 2-3). Although when code comes over the network, code fetching time becomes more significant, the combination of remote code fetching and dynamic recompilation is a powerful accelerator for distributed network applications.

5.3 (Virtual) Mobile Agents

Implementing agents 'roaming over' a set of servers is a simple and efficient high-level operation. First, we get the list of servers from the master server. Then we iterate through the appropriate remote-call negotiation with each site. Our agent's behavior is constrained through security and resource limitations of participating interpreters, having their own command filtering policies.

In particular, on a chat-only server, our roaming agent can only display a message. If the local interpreter allows gathering user information then our agent can collect it. If the local interpreter allows question/answering our agent will actually interact with the human user through the local server window.

Note that 'mobile agents' do not have to be implemented as code physically moved from one site to another. In this sense we can talk about *virtual mobile agents* which

are actually sets of synchronized remote predicate calls originating from a unique site, where most of the code is based/executed, while code actually moved can be kept to strict minimum, i.e. only a few remotely asserted clauses[2].

Our mobile agents are seen as *'connection brokers'* between participating independent server/client sites. For instance if two sites are willing to have a private conversation or code exchange they can do so by simply using the server/port/password information our agent can help them to exchange.

Note that full metaprogramming combined with source-level mobile-code have the potential of better security than byte-code level verification as it happens in Java, as meaningful analysis and verification of program properties is possible.

5.4 Crafting mobile agent scripts

To give a flavor of how agent scripts are crafted in our framework we will describe a *complete example* emphasizing the idea of Virtual Mobile Agents described in section 5.3. Each file begins with predicate `main/0` which is BinProlog's user-defined auto-start point.

The first file (script.pro) mostly dispatches Unix processes creating the other tasks in separate cmdtool windows (for demo purposes) of appropriate sizes.

Each file contains the code describing the behavior of an agent. The file master.pl initiates a local master server which keeps an updated list of all the existing servers and their properties (unique id, host, port).

```
main:-run_local_master_server.
```

The file agent.pl contains the code shared among different agents. `Main/0` starts by registering an unrestricted server, allowing execution of remote commands to the local master server. It is programmed in object oriented style with default actions to be performed by more specialized agents.

```
main:-
  default_master_server(_,P),
  master_server(localhost,P)=>>
    run_unrestricted_server.

init_super:-login(I),
  write(starting(I)),nl.

run_super:-login(Who),
  write(i_am(Who)),nl.

accept_mobile(GoalAndCode):-
   call(GoalAndCode).

reject_mobile(GoalAndCode):-
   write('mobile code not welcome!'),
   nl,write(GoalAndCode),nl.
```

The predicates for accepting/rejecting mobile 'visitors' can be used to enforce such policies on a server.

The first agent has default initialization `init/0` and its `run/1` action step set to reject visiting mobile agents.

```
:-[agent].
login('The First').
init:-init_super.

run(X):-run_super,reject_mobile(X).
```

Note the use of the builtin `login/1` predicate giving a unique identity to each agent.

The second agent is similar, except that its `run/1` action step is set to accept mobile agents.

The mobile agent itself uses BinProlog's built-in iteration over the servers registered on the master server as a powerful remote execution mechanism. In the presence of the `to_all/1` assumption, BinProlog's client code reacts by iterating over the set of servers matching a given pattern (everything in this case). This applies to every command, and in particular to remote execution requests specified by `do/1`.

```
main:-
  to_all(_)=>>animate(10).

master_server(localhost,9000).
do(X):-remote_run(X),!;true.

animate(Fuel):-do(init),
  mobile_code(Code),
  for(I,1,Fuel),
    do(run(Code)),
  fail ; write(done),nl.

mobile_code(
  (go:-write(visiting_you),nl)=>>go).
```

The predicate `animate/1` iterates over the set of servers, sending a small chunk of mobile code to be assumed on each server and then executed. Note that more sophisticated code migration schemes can be set up quite easily by assuming `code(SourceFile)`, which has the effect to lazily fetch to a given server the code needed for the execution of a remote query.

6 Related work

A very large number of research projects have recently started on mobile agent programming. Among the most

[2] Note however that suspending execution at one site, and then restarting it at another site can be done quite efficiently in BinProlog, where continuations are first order objects which can be put into a term to be sent over a socket, then read in and executed.

promising recent developments Luca Cardelli's Oblique project at Digital and mobile agent applications [1] and IBM Japan's aglets [5]. A growing number of sophisticated Web-based applications and tools are on the way to be implemented in LP/CLP languages. Among them, work with a similar emphasis on remote execution, agents, virtual worlds can be found in [6, 2, 14].

7 Conclusion and future work

Our remote execution mechanisms are based on a set of filtering interpreters which can be customized to support arbitrary negotiations with remote agents and are plugged in generic servers. The practical implementation is built on proven client/server technology, on top of a generic socket package, while giving the illusion of a 'Web of MOOs' with roaming mobile agents at the next level of abstraction.

A Java based Linda implementation, using a minimal set of logic programming components (unification, associative search) has been recently released (the Java TermServer, available at http://clement.info.umoncton.ca/ BinProlog). It allows to communicate bidirectionally with BinProlog, allowing creation of combined Java/Prolog mobile-agent programs. It holds promise for smooth cooperation with existing Java class hierarchies as well as various BinProlog based components with intelligence and flexible metaprogramming on the logic programming side combined with visualization and WWW programming abilities on the Java side.

Future work will focus on intelligent mobile agents integrating knowledge and controlled natural language processing abilities.

Acknowledgment

We thank for support from NSERC (grants OGP0107411 and 611024), and from the FESR of the Université de Moncton. Koen De Bosschere is research associate with the Fund for Scientific Research – Flanders.

References

[1] K. A. Bharat and L. Cardelli. Migratory applications. In *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology*, Nov. 1995. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-138.html.

[2] D. Cabeza and M. Hermenegildo. The Pillow/CIAO Library for Internet/WWW Programming using Computational Logic Systems. In Tarau et al. [13]. http://clement.info.umoncton.ca/ lpnet.

[3] N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, 1989.

[4] K. De Bosschere and P. Tarau. Blackboard-based Extensions in Prolog. *Software — Practice and Experience*, 26(1):49–69, Jan. 1996.

[5] IBM. Aglets. http://www.trl.ibm.co.jp/aglets.

[6] S. W. Loke and A. Davison. Logic programming with the world-wide web. In *Proceedings of the 7th ACM Conference on Hypertext*, pages 235–245. ACM Press, 1996.

[7] D. A. Miller. Lexical scoping as universal quantification. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 268–283, Cambridge, Massachusetts London, England, 1989. MIT Press.

[8] A. Mycroft and R. A. O'Keefe. A polimorphic type system for prolog. *Artificial Intelligence*, (23):295–307, 1984.

[9] P. Tarau. BinProlog 5.40 User Guide. Technical Report 97-1, Département d'Informatique, Université de Moncton, Apr. 1997. Available from *http://clement.info.umoncton.ca/BinProlog*.

[10] P. Tarau and V. Dahl. A Coordination Logic for Agent Programming in Virtual Worlds. In W. Conen and G. Neumann, editors, *Proceedings of Asian'96 Post-Conference Workshop on Coordination Technology for Collaborative Applications*, Singapore, Dec. 1996.

[11] P. Tarau, V. Dahl, and A. Fall. Backtrackable State with Linear Affine Implication and Assumption Grammars. In J. Jaffar and R. H. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security*, Lecture Notes in Computer Science 1179, pages 53–64, Singapore, Dec. 1996. Springer.

[12] P. Tarau, V. Dahl, S. Rochefort, and K. De Bosschere. LogiMOO: a Multi-User Virtual World with Agents and Natural Language Programming. In S. Pemberton, editor, *Proceedings of CHI'97*, pages 323–324, Mar. 1997. ACM ISBN 0-8979-926-2.

[13] P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors. *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP'96, Bonn, Sept. 1996. http://clement.info.umoncton.ca/ lpnet.

[14] P. Tarau and K. De Bosschere. Virtual World Brokerage with BinProlog and Netscape.' In Tarau et al. [13]. http://clement.info.umoncton.ca/ lpnet.