High-Level Networking with Mobile Code and First Order AND-Continuations *

PAUL TARAU and VERONICA DAHL†

▷ We describe a scheme for moving living code between a set of distributed processes coordinated with unification based Linda operations and its application to building a comprehensive Logic programming based Internet programming framework. *Mobile threads* are implemented by capturing first order continuations in a compact data structure sent over the network. Code is fetched lazily from its original base turned into a server as the continuation executes at the remote site. Our code migration techniques, in combination with a dynamic recompilation scheme ensure that heavily used code moves up smoothly on a speed hierarchy while volatile dynamic code is kept in a quickly updatable form. Among the examples, we describe how to build programmable client and server components (Web servers, in particular) and mobile agents.

Keywords: mobile computations, remote execution, networking, Internet programming, first order continuations, Linda coordination, blackboard-based logic programming, mobile agents, dynamic recompilation, code migration ◁

1. Introduction

Data mobility has been present since the beginning of networked computing, and is now used in numerous applications – from remote consultation of a database, to Web browsing.

*The authors thank for support from NSERC (grants OGP0107411 and 611024)

Address correspondence to Paul Tarau, Department of Computer Science University of North Texas P.O. Box 311366 Denton, Texas 76203 E-mail: *tarau@cs.unt.edu.*

†Logic and Functional Programming Group, Department of Computing Sciences, Simon Fraser University, E-mail: *veronica@cs.sfu.ca.*

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

Code mobility followed, often made transparent to users as with network files systems (i.e. Sun's NFS). Java's ability to execute applets directly in client browsers, can be seen as its most recent incarnation.

Migrating the state of the computation from one machine or process to another, still requires a separate set of tools. Java's remote method invocations (RMI) add *control mobility* and a (partially) automated form of *object mobility* i.e. integrated code (class) and data (state) mobility. The Oz 2.0 distributed programming proposal of [25] makes *object mobility* more transparent, although the mobile entity is still the state of the objects, not "live" code.

Mobility of "live code" is called *computation mobility* [4]. It requires interrupting execution, moving the state of a runtime system (stacks, for instance) from one site to another and then resuming execution. Clearly, for some languages, this can be hard or completely impossible to achieve.

Telescript and General Magic's new Odyssey [8] agent programming framework, IBM's Java based *aglets* as well as Luca Cardelli's Oblique [2] have pioneered implementation technologies achieving *computation mobility*.

This paper will show that we can achieve full *computation mobility* through our *mobile threads*, without needing a specially designed new language. They are implemented by a surprisingly small, source level modification of the BinProlog system, taking advantage of the availability of *first order continuations*[1] as well as of BinProlog's high level networking primitives. Mobile threads complete our Logic Programming based Internet programming infrastructure built in view of creating Prolog components which can interoperate with mainstream languages and programming environments. *Mobile threads* can be seen as a refinement of *mobile computations* as corresponding to *mobile partial computations* of any granularity. *Mobile agents* can be seen as a collection of synchronized *mobile threads* sharing common state [18]. We achieve synchronization using a variant of the Linda coordination protocol.

The paper is organized as follows:

- section 2 describes our networking infrastructure and Linda based client/server components
- Section 3 describes our code migration and code acceleration techniques (dynamic recompilation)
- Section 4 describes our mobile computation mechanism, as follows: subsection 4.2 intriduces engines and threads, subsection 4.3) reviews the underlying binarization mechanism used to implement our first order continuations, subsection 4.4 explains how we implement thread mobility by capturing continuations (subsection 4.4.1) and moving them from their base to their target (subsection 4.4.2), how this can be emulated with remote predicate calls (subsection 4.4.3) and how how mobile agents can be built within our framework (subsection 4.5)
- section 5 discusses related work
- section 6 presents our conclusions and future work

[1] I.e. continuations (representations of future computations) accessible as an ordinary data structure - a Prolog term in this case.

The main "paradigm independent" novelties of our contribution are:

- a technique, based on intuitionistic assumptions for dealing with complex networking code componentwise
- use of first order continuations for implementing *mobile computations*
- a flexible thread mobility algorithm expressed in terms of client-server role alternation and communication through Linda operations

Our contributions are synergetically integrated into a powerful agent building infrastructure, by putting together logic programming based knowledge processing, Linda-style coordination, and live code migration through mobile threads.

2. Basic Linda and Remote Execution Operations

2.1. Coordination of Linda clients

Our networking constructs are built on top of the popular Linda [5] coordination framework, enhanced with unification based pattern matching, remote execution and a set of simple client-server components merged together into a scalable peer-to-peer layer, forming a 'web of interconnected worlds':

```
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
```

The presence of the `all/2` collector avoids the need for backtracking over multiple remote answers. Note that the only blocking operation is `in/1`. Typically, distributed programming with Linda coordination follows consumer-producer patterns (see Fig. 1) with added flexibility over message-passing communication through associative search. Blocking `rd/1`, which waits until a matching term becomes available, without removing it, is easily emulated in terms of `in/1` and `out/1`, while non-blocking `rd/1` is emulated with `all/2`.

2.2. Remote Execution Mechanisms

Implementation of arbitrary remote execution is easy in a Linda + Prolog system, due to Prolog's *metaprogramming* abilities. No complex serialization/remote object packages are needed. Our primitive remote call operation is:

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

It implements deterministic *remote predicate calls* with (first)-answer or `'no'` returned to the calling site.

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 `=>>` [15, 6]:

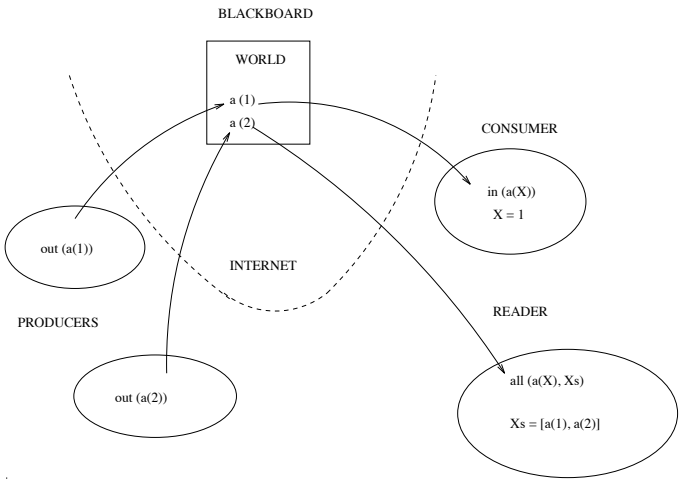

Figure 1. Basic Linda operations

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

Note that a `Channel` pattern is used to select a subset of relevant servers, and in particular, when `Channel` is a "match all" free logical variable, all of them. By using term subsumption this allows building sophisticated "publish/subscribe" communication patterns hierarchies.

2.3. Servants: basic Linda agents

A *servant* is one of the simplest possible *agents*, which pulls commands from a server and runs them locally:

```
servant:-
  in(todo(Task)),
  call(Task),
  servant.
```

Note that *servant* is started as a background thread. No 'busy wait' is involved, as the servant's thread blocks until `in(todo(Task))` succeeds. More generally, distributed event processing is implemented by creating a 'watching' agent attached to a thread for each pattern.

As *servants* pulling commands are operationally indistinguishable from *servers* acting upon clients' requests, they can be used as emulators for *servers*. A class of obvious applications of this ability is their use as pseudo-servers running on machines with dynamically allocated IP addresses (as offered by most ISP today), laying behind firewalls. This mechanism also works when, because of security restrictions, server components cannot be reached from outside, as in the case of *Java applets* which cannot listen on ports of the client side machine.

2.4. Server side code

Servants as well as other clients can connect to BinProlog *servers*. Higher order *call/N* [13], combined with intuitionistic assumptions ``=>>'', are used to pass arbitrary *interactors* to generic server code:

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

server_loop:-
  repeat,
    interact,
  assumed(server_done),
  !.

interact:-
  assumed(Interactor),
  assumed(Server),
  % 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*.

The components of a 'generic' default server can be overridden through the use of *intuitionistic implication* to obtain customized special purpose servers. The use of intuitionistic implications (pioneered by Miller's work [12]) 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 [15, 21, 6] for more information on assumptions and their applications.

2.5. Modular HTTP server component building: do or delegate

We will show in this section a typical application of our component based server building technology: how to enhance efficiently the HTTP protocol, to handle server side Prolog scripts directly, without using GGIs or vendor specific server side extensions.

The top goal of the HTTP server looks as follows:

```
run_http_server:-
  server_action(http_server_action)=>>run_server.

http_server_action(ServiceSocket):-
  socket(ServiceSocket)=>>http_server_step(ServiceSocket).
```

The `http_server_action` is passed to the inner server loop using intuitionistic implication. This allows reusing general server logic, independently of a particular protocol. The action itself is described in `http_server_step`: it consists of preparing a fall-back mechanism to a standard HTTP server, unless the request is for a file recognized as Prolog code, using the *redirection* facilities built in the HTTP protocol.

```
http_server_step(Socket):-
  ( assumed(fallback_server(FallBackServer))->true
  ; FallBackServer="http://localhost:80"
  ),
  server_try(Socket,sock_readln(Socket,Question)),
  http_get_client_header(Socket,Header),
  http_process_query(Socket,Question,Header,FallBackServer).
```

Our very simple query processor uses Assumption Grammars [6]. Their ability to handle multiple DCG streams [21] are instrumental, as we use more than one independent grammar processor in the process.

```
http_process_query(Socket,Qs,Css,FallBackServer):-
  #<Qs, % sets input string from grammar
  match_word("GET "),
  match_before(" ",PathFile,_),
  match_word("HTTP/"),
  #>_version, %
  !,
  split_path_file(PathFile,Ds,Fs),
  !,
  has_text_file_sufix(Fs,Suf),
  !,
  ( Suf=".pro"->http_process_local(Socket,Ds,Fs,Suf,Css)
  ; write('redirecting '),write_chars(Ds),write_chars(Fs),nl,
    http_send_line(Socket,"HTTP/1.0 302 Found"),
    make_cmd0(["Location: ",FallBackServer,Ds,Fs],Redirect),
    http_send_line(Socket,Redirect),
    http_send_line(Socket,"")
  ),
  close_socket(Socket).
```

Our HTTP server component fits in 76 lines of code and can be used to set up Prolog based Web processing by simply staring it in a command window on any Unix machine or PC, with no execution overhead, as in the case of with CGI scripts. It basically offers the advantages of server side includes (SSIs) without requiring integration into a server, often subject to using the languages the server supports.

2.6. Master Servers: Connecting a Web of Virtual Places

A *virtual place* is implemented as a server listening on a port which can spawn clients in the same or separate threads interacting with other servers.

A master server on a 'well-known' host/port is used as a registration service to exchange identification information among peers composed of clients and a server, usually running as threads of the same process.

As in the case of the HTTP server we can derive a master server by specializing its interactor components through intuitionistic implications.

3. Code migration and code acceleration techniques

We have seen that setting up a self contained networking infrastructure (Web protocols included) is fairly simple. The next step is emphasizing on mobile agent support, particularly promising in synergy with knowledge processing capabilities - a key strength of Logic Programming systems.

3.1. Lazy code fetching

In BinProlog, code is fetched lazily over the network, one predicate at a time, as needed by the execution flow.

Code 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.

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, at predicate level (predicates are cached as dynamic code on the server to efficiently serve multiple clients).

3.2. 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).

Our implementation of dynamic recompilation for BinProlog is largely motivated by the difficulty/complexity of relying on the programmer to specify execution methods for remote code.

The intuition behind the dynamic recompilation algorithm of BinProlog is that *update* vs. *call* based *statistics* are associated to each predicate declared or detected as dynamic. Dynamic (re)compilation is triggered for relatively non-volatile predicates, which are promoted on the *'speed-hierarchy'* to a faster implementation method (interpreted `->` bytecode `->` native). The process is restarted from the 'easier to change' interpreted representation, kept in memory in a compact form, upon an update.

We can describe BinProlog's dynamic *'recompilation triggering statistics'* through a simple 'thermostat' metaphor. *Updates* (assert/retract) to a predicate have the effect of increasing its associated 'temperature', while *Calls* will decrease it. Non-volatile ('cool') predicates are dynamically recompiled, while recompilation is

avoided for volatile ('hot') predicates. A *ratio* based on cooling factors (number of calls, compiled/interpreted execution speed-up etc.) and heating factors (re-compilation time, number of updates etc.) smoothly adjusts for optimal overall performance, usually within a factor of 2 from static code.

4. Computation Mobility with Threads and Continuations

4.1. Why computations need to be mobile?

Advanced *mobile object* and *mobile agents* agent systems have been built on top of Java's dynamic class loading and its new reflection and remote method invocation classes. IBM Japan's Aglets or General Magic's Odyssey provide comprehensive mobility of code and data. Moreover, data is encapsulated as state of objects. This property allows protecting sensitive components of it more easily. Distributed Oz 2 provides fully transparent movement of objects over the network, giving the illusion that the same program runs on all the computers.

So *why do we need* the apparently more powerful concept of mobile "live code" i.e. mobile execution state?

Our answer to this question is that live mobile code is needed because is still *semantically simpler* than mobile object schemes. Basically, all that a programmer needs to know is that his or her program has moved to a new site and it is executing there. A unique (in our case `move_thread`) primitive, with an intuitive semantics, needs to be learned. When judging about how appropriate a language feature is, we think that the way it looks to the end user is among the most important ones. For this reason, mobile threads are competitive with sophisticated *object mobility* constructs on "end-user ergonomy" grounds, while being fairly simple to implement, as we have shown, in languages in which continuations can be easily represented as data structures.

And *what if the host language does not offer first order continuations*? A simple way around this is to implement on top of it a script interpreter (e.g. a subset of Scheme or Prolog) which does support them! As it is a good idea to limit code migration to lightweight scripts anyway, this is a very practical solution for either C/C++ or Java based mobile code solutions, without requiring class-specific serialization mechanisms.

4.2. Engines and Answer Threads

Mobile computations really make sense only when multiple computation threads can coexist at a given place. We build this infrastructure in two steps: an engine encapsulating the state of an independent computation and a thread actually running it. Note that engines can also be used without multi-threading, as a form of coroutining.

4.2.1. Engines BinProlog allows launching multiple Prolog engines having their own stack groups (heap, local stack and trail). An engine can be seen as an abstract data-type which produces a (possibly infinite) stream of solutions as needed. To create a new engine, we use:

```
create_engine(+HeapSize,+StackSize,+TrailSize,-Handle)
```

or, by using default parameters for the stacks:

```
create_engine(-Handle)
```

The `Handle` is a unique integer denoting the engine for further processing. To 'fuel' the engine with a goal and an expected answer variable we use:

```
load_engine(+Handle,+Goal,+AnswerVariable)
```

No processing, except the initialization of the engine takes place, and no answer is returned with this operation.

To get an answer from the engine we use:

```
ask_engine(+Handle,-Answer)
```

Each engine has its own heap garbage collection process and backtracks independently using its choice-point stack and trail during the computation of an answer. Once computed, an answer is copied from an engine to its "master".

When the stream of answers reaches its end, `ask_engine/2` will simply fail. The resolution process in an engine can be discarded at any time by simply loading another goal with `load_engine/3`. This allows avoiding the cost of backtracking, for instance in the case when a single answer is needed, as well as garbage collection costs.

If for some reason we are not interested in the engine any more, we can free the space allocated to the engine and completely discard it with:

```
destroy_engine(+Handle)
```

The following example [2] in the BinProlog distribution [15] shows a sequence of the previously described operations:

```
?-create_engine(E),
  load_engine(E,append(As,Bs,[1,2]),As+Bs),
  ask_engine(E,R1),write(R1),nl,
  ask_engine(E,R2),write(R2),nl,
  destroy_engine(E).
```

Multiple 'orthogonal engines' as shown in Figure 1 enhance the expressiveness of Prolog by allowing an AND-branch of an engine to collect answers from multiple OR-branches of another engine. They give to the programmer the means to see as an abstract sequence and control, the answers produced by an engine, in a way similar to Java's `Enumeration` interface.

4.2.2. Threads Engines can be assigned to their own thread by using BinProlog's thread package (currently fully implemented on win32 platform). A unique primitive is needed,

```
ask_thread(E,R)
```

which launches a new thread `R` to perform the computation of an answer of engine `E`. On top of this facility each thread can implement a separate server, client or become the base of a mobile agent.

Thread synchronization is provided through monitor objects, handled with:

[2] See more in files library/engines.pl, progs/engtest.pl

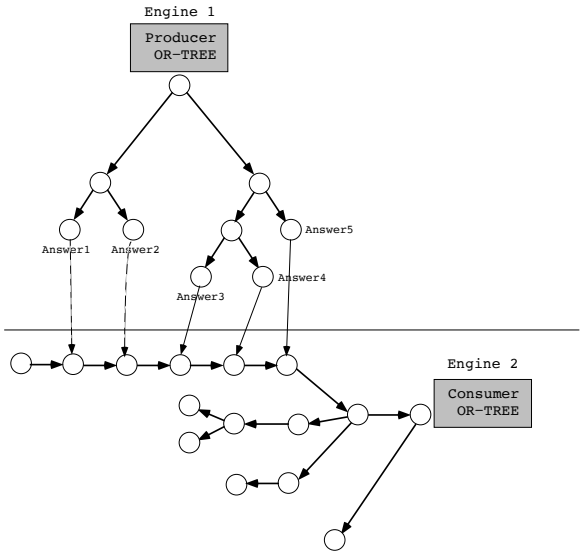

Figure 1. Orthogonal Engines

```
synchronize_on(Monitor,Goal,Answer)
```

The thread waits until the monitor is free, executes Goal, frees the monitor and returns Answer.

The thread attached to an engine can be obtained with:

```
get_engine_thread(Engine,Thread)
```

and can be controlled directly with:

```
thread_suspend(Thread) % suspends the thread
thread_resume(Thread)  % resumes a suspended thread
thread_join(Thread)    % waits until a thread terminates
thread_cancel(Thread) % discards a thread
```

4.3. First order Continuations through Binarization

Having first order continuations largely simplifies implementation of mobile code operations - a thread is suspended, its continuation is packed, sent over the network and resumed at a different place.

We will shortly explain here BinProlog's continuation passing preprocessing technique, which results in availability of continuations as data structures accessible to the programmer.

The binarization transformation Binary clauses have only one atom in the body (except for some in-line 'builtin' operations like arithmetics), and therefore they need no 'return' after a call. A transformation introduced in [17] allows to faithfully represent logic programs with operationally equivalent binary programs.

To keep things simple, we will describe our transformations in the case of definite programs. We will follow here the notations of [23].

Let us define the *composition* operator $\oplus$ that combines clauses by unfolding the leftmost body-goal of the first argument.

Let $\texttt{A}_0\texttt{:-A}_1\texttt{,A}_2\texttt{,...,A}_n$ and $\texttt{B}_0\texttt{:-B}_1\texttt{,...,B}_m$ be two clauses (suppose $n > 0, m \geq 0$). We define

$$(\texttt{A}_0\texttt{:-A}_1\texttt{,A}_2\texttt{,...,A}_n) \oplus (\texttt{B}_0\texttt{:-B}_1\texttt{,...,B}_m) = (\texttt{A}_0\texttt{:-B}_1\texttt{,...,B}_m\texttt{,A}_2\texttt{,...,A}_n)\theta$$

with $\theta = \text{mgu}(\texttt{A}_1,\texttt{B}_0)$. If the atoms $\texttt{A}_1$ and $\texttt{B}_0$ do not unify, the result of the composition is denoted as $\bot$. Furthermore, as usual, we consider $\texttt{A}_0\texttt{:-true,A}_2\texttt{,...,A}_n$ to be equivalent to $\texttt{A}_0\texttt{:-A}_2\texttt{,...,A}_n$, and for any clause C, $\bot \oplus \texttt{C} = \texttt{C} \oplus \bot = \bot$. We assume that at least one operand has been renamed to a variant with variables standardized apart.

This Prolog-like inference rule is called LD-resolution and it has the advantage of giving a more accurate description of Prolog's operational semantics than SLD-resolution. Before introducing the binarization transformation, we describe two auxiliary transformations.

The first transformation converts facts into rules by giving them the atom `true` as body. E.g., the fact `p` is transformed into the rule `p :- true`.

The second transformation, inspired by [27], eliminates the metavariables by wrapping them in a `call/1` goal. E.g., the rule `and(X,Y):-X, Y` is transformed into `and(X,Y) :- call(X), call(Y).`

The transformation of [17] (*binarization*) adds continuations as extra arguments of atoms in a way that preserves also first argument indexing.

Let P be a definite program and $Cont$ a new variable. Let T and $E = p(T_1, ..., T_n)$ be two expressions.[3] We denote by $\psi(E, T)$ the expression $p(T_1, ..., T_n, T)$. Starting with the clause

(C) $\quad A : -B_1, B_2, ..., B_n.$

we construct the clause

(C') $\quad \psi(A, Cont) : -\psi(B_1, \psi(B_2, ..., \psi(B_n, Cont))).$

The set P' of all clauses C' obtained from the clauses of P is called the binarization of P.

The following example shows the result of this transformation on the well-known 'naive reverse' program:

```
app([],Ys,Ys,Cont):-true(Cont).
app([A|Xs],Ys,[A|Zs],Cont):-
  app(Xs,Ys,Zs,Cont).

nrev([],[],Cont):-true(Cont).
nrev([X|Xs],Zs,Cont):-
  nrev(Xs,Ys,app(Ys,[X],Zs,Cont)).
```

The transformation preserves a strong operational equivalence with the original program with respect to the LD resolution rule, which is *reified* in the syntactical structure of the resulting program, i.e. each resolution step of an LD derivation on a definite program P can be mapped to an SLD-resolution step of the binarized program P'.

[3] Atom or term.

Clearly, continuations become explicit in the binary version of the program. We have devised a technique to access and manipulate them in an intuitive way, by modifying BinProlog's binarization preprocessor. Basically, the clauses constructed with `::-` instead of `:-` are considered as being already in binary form, and not subject therefore to further binarization. By explicitly accessing their arguments, a programmer is able to access and modify the current continuation as a 'first order object'. Note however that code *referring* to the continuation is also *part* of it, so that some care should be taken in manipulating the circular term representing the continuation from 'inside'.

4.4. Mobile threads: Take the Future *and Run*

As continuations (describing *future* computations to be performed at a given point) are first order objects in BinProlog, it is easy to extract from them a conjunction of goals representing *future* computations intended to be performed at another site, send it over the network and resume working on it at that site. The natural unit of mobility is a *thread* moving to a server executing multiple local and remotely originated threads. *Threads communicate with their local and remote counterparts, listening on ports through the Linda protocol, as described in [7].* This combination of Linda based coordination and thread mobility is intended to make building complex, pattern based agent scripts fairly easy.

4.4.1. Capturing continuations Before moving to another site, the current continuation needs to be captured in a data structure (see Appendix I). For flexibility, a wrapper `capture_cont_for/1` is used first to restrict the scope of the continuation to a (deterministic) toplevel `Goal`. This avoids taking irrelevant parts of the continuation (like prompting the user for the next query) to the remote site inadvertently.

A unique logical variable is used through a backtrackable linear assumption `cont_marker(End)` to mark the end of the scope of the continuation with `end_cont(End)`.

From inside the continuation, `call_with_cont/1` is used to extract the relevant segment of the continuation. Towards this end, `consume_cont(Closure,Marker)` extracts a conjunction of goals from the current continuation until Marker is reached, and then it applies `Closure` to this conjunction (calls it with the conjunction passed to `Closure` as an argument).

Extracting the continuation itself is easy, by using BinProlog's ability to accept user defined binarized clauses (introduced with ::- instead of :-), accessing the continuation as a 'first order' object:

```
get_cont(Cont,Cont)::-true(Cont).
```

4.4.2. The Continuation Moving Protocol Our continuation moving protocol can be described easily in terms of synchronized *source side*[4], and *target side* operations.

[4]which will be also shortly called the *base* of the mobile thread

Source side operations

- wrap a Goal with a unique terminator marking the end of the continuation to be captured, and call it with the current continuation available to it through a linearly assumed fact[5]
- reserve a free port P for the future code server
- schedule on the target server a sequence of actions which will lead to resuming the execution from right after the `move_thread` operation (see target side operations), return and become a code server allowing the mobile thread to fetch required predicates one a time

Target side operations are scheduled as a sequence of goals extracted from the current continuation at the *source side* , and received over the network together with a small set of synchronization commands:

- schedule as delayed task a sequence of goals received from the source side and return
- wait until the *source side* is in server mode
- set up the back links to the source side as assumptions
- execute the delayed operations representing the moved continuation
- fetch code from the source side as needed for execution of the goals of the moved continuations and their subcalls
- shut down the code server on the source side

Communication between the base and the target side is done with *remote predicate calls* protected with *dynamically generated passwords* shared between the two sides before the migratory component "takes off".

Initially the target side waits in server mode. Once the continuation is received on the target side, the source side switches in server mode ready to execute code fetching and persistent database update requests from its mobile counterpart on the target side.

Fig. 2 shows the connections between a mobile thread and its base.

Note that when the base turns into a server, it offers its *own code* for remote use by the moved thread - a kind of virtual "on demand" process cloning operation, one step at a time. As the server actually acts as a code cache, multiple moving threads can benefit from this operation. Note also that only predicates needed for the migratory segment of the continuation are fetched. This ensures that migratory code is kept lightweight for most mobile applications. Synchronized communication, using Linda operations can occur between the mobile thread and its base server, and through the server, between multiple mobile threads which have migrated to various places.

As our networking infrastructure, our *mobile threads* are platform independent. Like Java, BinProlog [16] is a platform independent emulator based language. As

[5] BinProlog's linear assumptions are backtrackable additions to the database, usable at most once.

a consequence, a thread can start on a Unix machine and move transparently to a Windows NT system and back. For faster, platform specific execution, BinProlog provides compilation to C of static code using an original partial translation technique described in [24].

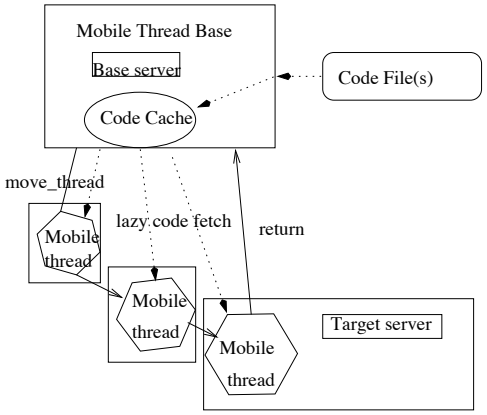

Figure 2. Launching a mobile thread from its base

4.4.3. Emulating computation mobility through control mobility As shown in [20], part of the functionality of *mobile computations* can be emulated in terms of remote predicate calls combined with remote code fetching. An implicit *virtual place* (host+port) can be set as the target of the remote calls. Then, it is enough to send the top-level goal to the remote side and have it fetch the code as needed from a server at the site from where the code originates.

Note however that this is less efficient in terms of network transactions and less reliable than sending the full continuation at once as with our *mobile threads*.

4.5. Mobile Agents

Mobile agents can be seen as a collection of synchronized *mobile threads* sharing common state [18].

Mobile agents are implemented by iterating *thread mobility* over a set of servers[6] known to a given master server. An efficient pyramidal deployment strategy can be used to efficiently implement, for instance, *push technology* through mobile agents. Inter-agent communication can be achieved either by rendez-vous of two mobile threads at a given site, by communicating through a local Prolog database, or through the base server known to all the deployed agents. Communication with the base server is easily achieved through remote predicate calls with `remote_run`. Basic security of mobile agents is achieved with randomly generated passwords, required for `remote_run` operations, and by running them on a restricted BinProlog machine, without user-level file write and external process spawn operations.

[6] possibly filtered down to a relevant subset using a 'channel'-like pattern

5. Related work

Remote execution and code migration techniques are pioneered by [1, 10, 14]. Support for remote procedure calls (RPC) are part of major operating systems like Sun's Solaris and Microsoft's Windows NT.

A very large number of research projects have recently started on mobile computations/mobile agent programming. Among the pioneers, Kahn and Cerf's Knowbots [11]. Among the most promising recent developments, Luca Cardelli's Oblique project at Digital and mobile agent applications [2] and IBM Japan's aglets [9]. Mobile code technologies are pioneered by General Magic's Telescript (see [8] for their last Java based *mobile agent* product). General Magic's Portico software combines mobile code technologies and voice recognition based command language (MagicTalk). Another mobility framework, sharing some of our objectives towards transparent high level distributed programming is built on top of Distributed Oz [25, 26], a multi-paradigm language, also including a logic programming component. Although thread mobility is not implemented in Distributed Oz 2, some of this functionality can be emulated in terms of network transparent mobile objects. Achieving the illusion of a unique application transparently running on multiple sites makes implementing shared multi-user applications particularly easy. We can emulate this by implementing mobile agents (e.g. avatars) as mobile threads with parts of the shared world *visible* to an agent represented as dynamic facts, lazily replicated through our lazy code fetching scheme when the agent moves. Both Distributed Oz 2 and our BinProlog based infrastructure need a full language processor (Oz 2 or BinProlog) to be deployed at each node. However, assuming that a Java processor is already installed, our framework's Java client (see [20, 19]) allows this functionality to be available through applets attached to a server side BinProlog thread. A calculus of *mobility* dealing with containers, called *ambients*, is described in [3]. The calculus covers at very high level of generality movement and permissions to move from one ambient to another and show how fundamental computational mechanisms like Turing machines as well as process calculi can be expressed within the formalism. Our *coordination logic* of [18] introduces similar concepts, based on programming mobile avatars in shared virtual worlds. Two classes of containers, *clonable* and *unique* regulate creation of new instances (clones) and non-copiable (unique) entities (like electronic money), as well as their movement.

6. Conclusion

We have described how mobile threads are implemented by capturing first order continuations in a data structure sent over the network. Supported by *lazy code fetching* and *dynamic recompilation*, they have been shown to be an effective framework for implementing mobile agents.

The techniques presented here are not (Bin)Prolog specific. The most obvious porting target of our design is to functional languages featuring first order continuations and threads. Another porting target is Java and similar object oriented languages having threads, reflection classes and remote method invocation. We are working on a Java based component using an embedded continuation passing Prolog interpreter which is already able to interoperate with BinProlog (latest version at urlhttp://www.binnetcorp.com/Jinni). An interesting application is using Bin-

Prolog as an accelerator for Java based threads through migration to BinProlog, execution of a computationally intensive task and return to the Java component.

Future work will focus on intelligent mobile agents integrating knowledge and controlled natural language processing abilities, following our previous work described in [22].

REFERENCES

1. G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden System: A Technical Review. *IEEE Transactions on Software Engineering*, 11(1):43–59, January 1985.

2. 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.

3. L. Cardelli. Mobile ambients. Technical report, Digital, 1997. http://www.research.digital.com/ SRC/personal/Luca_Cardelli/Papers.html.

4. L. Cardelli. Mobile Computation. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems - Towards the Programmable Internet*, pages 3–6. Springer-Verlag, LNCS 1228, 1997.

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

6. V. Dahl, P. Tarau, and R. Li. Assumption Grammars for Processing Natural Language. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 256–270, MIT press, 1997.

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

8. GeneralMagicInc. Odissey. 1997. available at http://www.genmagic.com/agents.

9. IBM. Aglets. http://www.trl.ibm.co.jp/aglets.

10. E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

11. R. E. Kahn and V. G. Cerf. The digital library project, volume i: The world of knowbots. 1988. Unpublished manuscript, Corporation for National Research Initiatives, Reston, Va., Mar.

12. 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.

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

14. J. W. Stamos and D. K. Gifford. Remote Evaluation. *ACM Transaction on Programming Languages and Systems*, 12(4):537–565, October 1990.

15. P. Tarau. BinProlog 5.75 User Guide. Technical Report 97-1, Département d'Informatique, Université de Moncton, Apr. 1997. Available from *http://clement.info.umoncton.ca/BinProlog*.

16. P. Tarau. BinProlog 7.0 Professional Edition: User Guide. Technical report, BinNet Corp., 1998. Available from http://www.binnetcorp.com/BinProlog.

17. P. Tarau and M. Boyer. Elementary Logic Programs. In P. Deransart and J. Maluszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 159–173. Springer, Aug. 1990.

18. 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.

19. P. Tarau, V. Dahl, and K. De Bosschere. A Logic Programming Infrastructure for Remote Execution, Mobile Code and Agents. In *Proceedings of WETICE'97*, pages 106–112, IEEE Computer Society Press, June 1997.

20. P. Tarau, V. Dahl, and K. De Bosschere. Logic Programming Tools for Remote Execution, Mobile Code and Agents. In *Proceedings of ICLP'97 Workshop on Logic Programming and Multi Agent Systems*, Leuven, Belgium, July 1997.

21. 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".

22. 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.

23. P. Tarau and K. De Bosschere. Memoing with Abstract Answers and Delphi Lemmas. In Y. Deville, editor, *Logic Program Synthesis and Transformation*, Springer-Verlag, pages 196–209, Louvain-la-Neuve, July 1993.

24. P. Tarau, K. De Bosschere, and B. Demoen. Partial Translation: Towards a Portable and Efficient Prolog Implementation Technology. *Journal of Logic Programming*, 29(1–3):65–83, Nov. 1996.

25. P. Van Roy, S. Haridi, and P. Brand. Using mobility to make transparent distribution practical. 1997. manuscript.

26. P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhouer. Mobile Objects in Distributed Oz. *ACM TOPLAS*, 1997. to appear.

27. D. H. D. Warren. Higher-order extensions to Prolog – are they needed? In D. Michie, J. Hayes, and Y. H. Pao, editors, *Machine Intelligence 10*. Ellis Horwood, 1981.

Appendix I: Capturing First Order Continuations in BinProlog

```
% calls Goal with current continuation available to its inner calls
capture_cont_for(Goal):-
  assumeal(cont_marker(End)),
    Goal,
  end_cont(End).

% passes Closure to be called on accumulated continuation
call_with_cont(Closure):-
  assumed(cont_marker(End)),
  consume_cont(Closure,End).

% gathers in conjunction goals from the current continuation
% until Marker is reached when it calls Closure on it
consume_cont(Closure,Marker):-
  get_cont(Cont),
  consume_cont1(Marker,(_,_,_,Cs),Cont,NewCont), % first _
  call(Closure,Cs),                              % second _
  % sets current continuation to leftover NewCont
  call_cont(NewCont).                            % third _

% gathers goals in Gs until Marker is hit in continuation Cont
% when leftover LastCont continuation (stripped of Gs) is returned
consume_cont1(Marker,Gs,Cont,LastCont):-
   strip_cont(Cont,Goal,NextCont),
   ( NextCont==true-> !,errmes(in_consume_cont,expected_marker(Marker))
   ; arg(1,NextCont,X),Marker==X->
     Gs=Goal,arg(2,NextCont,LastCont)
   ; Gs=(Goal,OtherGs),
     consume_cont1(Marker,OtherGs,NextCont,LastCont)
   ).

% this 'binarized clause' gets the current continuation
get_cont(Cont,Cont)::-true(Cont).

% sets calls NewCont as continuation to be called next
call_cont(NewCont,_) ::- true(NewCont).
```

Appendix II: Thread Mobility in BinProlog

```
% wraps continuation of current thread to be taken
% by inner move_thread goal to be executed remotely
wrap_thread(Goal):-
  capture_cont_for(Goal).

% picks up wrapped continuation,
% jumps to default remote site and runs it there
move_thread:-
  call_with_cont(move_with_cont).

% moves to remote site goals Gs in current continuation
move_with_cont(Gs):-
  % gets info about this host
  detect_host(BackHost),
  get_free_port(BackPort),
  default_password(BackPasswd),
  default_code(BackCode),
  % runs delayed remote command (assumes is with +/1)
  remote_run(
     +todo(
       host(BackHost)=>>port(BackPort)=>>code(BackCode)=>>(
         sleep(5), % waits until server on BackPort is up
         % runs foals Gs picked up from current continuation
         (Gs->true;true), % ignores failure
         % stops server back on site of origin
         stop_server(BackPasswd)
       )
     )
  ),
  % becomes data and code server for mobile code until is
  % stopped by mobile code possessing password
  server_port(BackPort)=>>run_unrestricted_server.
```