Mobile Threads through First Order Continuations

Paul Tarau[1] and Veronica Dahl[2]

[1] Université de Moncton
Département d'Informatique
Moncton, N.B. Canada E1A 3E9,
tarau@info.umoncton.ca
[2] Logic and Functional Programming Group
Department of Computing Sciences
Simon Fraser University
Burnaby, B.C. Canada V5A 1S6
veronica@cs.sfu.ca

Abstract. 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 techniques, in combination with a dynamic recompilation scheme ensuring that heavily used code moves up smoothly on a speed hierarchy, are shown to be an effective means for implementing mobile agents.
Keywords: mobile computations, remote execution, metaprogramming, first order continuations, Linda coordination, blackboard-based logic programming, mobile code, mobile agents

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.

Code mobility followed, often in disguised form, with human hand waiving being part of the *workflow*, as is still the case with ftp-transferred, untarred-uncompressed and makefile-hacked Unix installs. More transparently, executables coming over network file systems on intranets as well as self-installing Windows programs are all instances of mobile code. Self-updating software is probably the maximum of functionality which can be expressed in this framework. The most well-known example of code mobility is Java's ability to execute applets directly in client browsers in a fairly secure way through dynamic class fetching and bytecode verification.

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* [3]. 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 [9] agent programming framework, IBM's Java based *aglets* as well as Luca Cardelli's Oblique [1] have pioneered implementation technologies achieving *computation mobility*.

Towards the same objective, this paper will show that we can achieve full *computation mobility* through our *mobile threads*. 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 [16].

The paper is organized as follows:

- section 2 describes our networking infrastructure and Linda based client/server components
- section 3 introduces code mobility primitives
- section 4 describes engines and threads
- section 5 explains how continuations are represented as data structures
- section 6 explains how we implement thread mobility by capturing continuations (subsection 6.1) and by moving them from their base to their target (subsection 6.2)
- section 8 describes how mobile agents can be built with our framework
- section 9 discusses related work
- section 10 presents our conclusions and future work

Our Internet programming infrastructure and in particular the technologies we have designed and implemented for code and control mobility are new in the logic programming domain. The main "paradigm independent" novelties of our contribution are:

- use of first order continuations for implementing *mobile computations*
- expressing thread mobility in terms of client-server role alternation

[1] I.e. continuations accessible as an ordinary data structure - a Prolog term in this case.

2 A Logic Programming based Internet Programming Infrastructure

We refer to [17] for the details of our high-level client-server programming primitives and security issues and to [14] for the platform independent and Java-compatible socket-level primitives of BinProlog.

Our networking constructs are built on the top of the popular Linda [4] coordination framework, enhanced with unification based pattern matching, remote execution and a set of simple client-server components melted 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
remote_run(Goal): starts a thread executing Goal on server
stop_server: stops the server
```

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

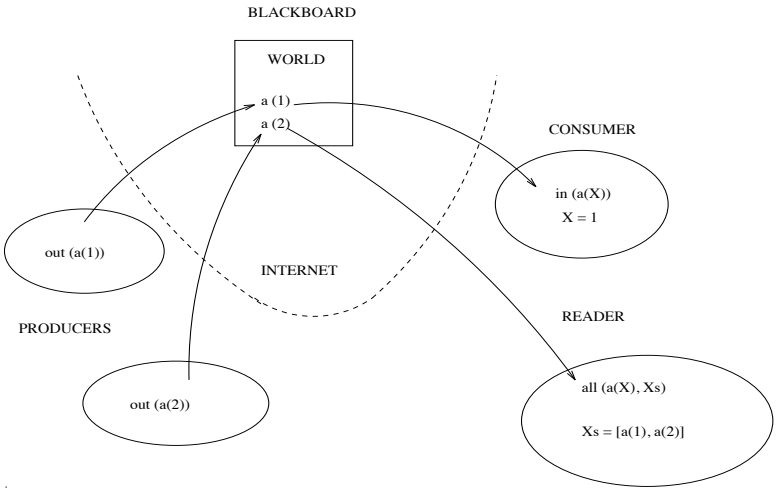

Fig. 1. Basic Linda operations

2.1 A Servant: a simple Linda based BinProlog agent

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.

Interaction with Java Applets. To avoid Java security restrictions, an applet emulates a *BinProlog servant*, using an interface module written in Java called a *Linda Interactor*. BinProlog communicates with this special purpose trimmed down pure Prolog engine which supports the same unification based Linda protocol as BinProlog and acts as a gateway between knowledge processing components written in Prolog and visual and reactive components written in Java. In particular, this design allows Java applets to cooperate with the rest of our 'peer-to-peer' network of BinProlog interactors, as the Java servant simply pulls out commands from a proxy server on the site where the applet originates from.

2.2 Server side code

Servants as well as other clients can connect to BinProlog *servers*. Higher order *call/N* [12], 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 [11]) 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 [14,20,6] for more information on assumptions and their applications.

2.3 Master Servers: Connecting a Web of Worlds

The MOO[2] inspired 'web of worlds' metaphor [21,23] implemented as as a set of BinProlog and Java based 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.

A *virtual place* (world) is implemented as a server listening on a port which can spawn clients in the same or separate threads interacting with other servers through a simple question/answer protocol.

A master server on a 'well-known' host/port is used to exchange identification information among peers composed of clients and a server (Fig. 2), usually running as threads of the same process.

2.4 Remote execution mechanisms

Implementation of arbitrary remote execution is easy in a Linda + Prolog system, due to Prolog's *metaprogramming* abilities. No complex serialization/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:

[2] 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

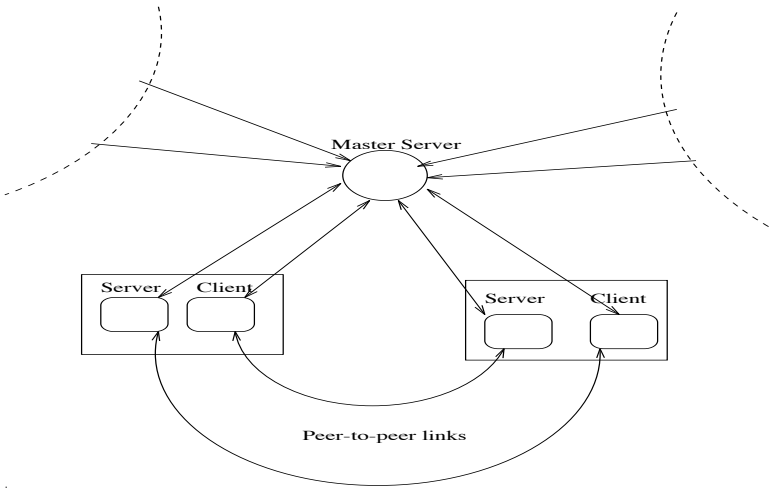

Fig. 2. A Web of Worlds

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

3 Mobile Code

We will shortly discuss here the basic Mobile Code facilities we have implemented.

3.1 Lazy code fetching

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

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).

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 lazy remote code fetching and dynamic recompilation is a powerful accelerator for distributed network applications.

3.2 Dynamic recompilation

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 (recompilation time, number of updates etc.) smoothly adjusts for optimal overall performance, usually within a factor of 2 from static code.

4 Engines and Answer Threads

4.1 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 an new engine, we use:

```
% :-mode create_engine(+,+,+,-).
create_engine(HeapSize,StackSize,TrailSize,Handle)
```

or, by using default parameters for the stacks:

```
% :-mode create_engine(-).
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:

```
% :-mode load_engine(+,+,+).
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:

```
% :-mode ask_engine(+,-).
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:

```
% :-mode destroy_engine(+).
destroy_engine(Handle)
```

The following example (see more in files `library/engines, progs/engtest.pl` in the BinProlog distribution [14]) shows a sequence of the previously described operations:

```
?-create_engine(E),
  load_engine(E,append(As,Bs,[A,B,B,A]),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 3 enhance the expressiveness of Prolog by allowing an AND-branch of an engine to collect answers from multiple OR-branch 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.

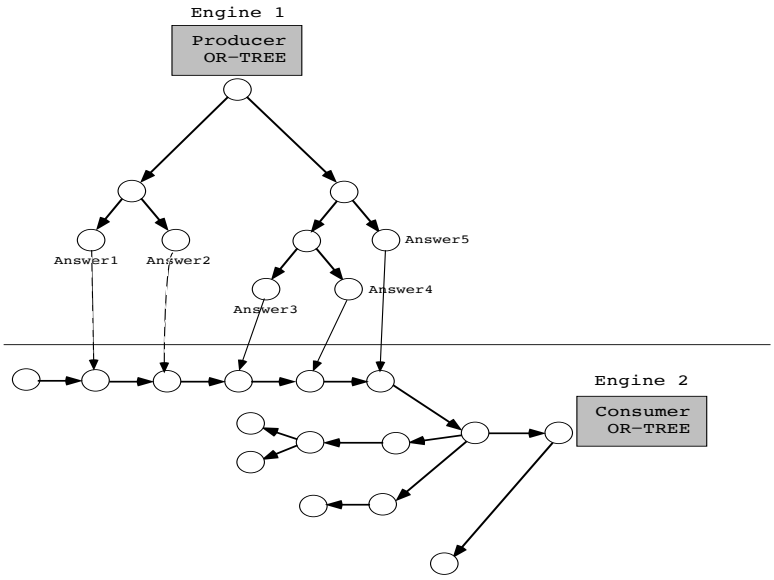

Fig. 3. Orthogonal Engines

4.2 Threads

Engines can be assigned to their own thread by using BinProlog's POSIX thread package. 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.

5 First order Continuations through Binarization

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 [15] 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 [22].

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 [15] (*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)).
```

These transformations preserve 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', in the sense that if G is an atomic goal and

[3] Atom or term.

$G' = \psi(G, true)$, then computed answers obtained querying P with G are the same as those obtained by querying P' with G'.

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'.

6 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].

6.1 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 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).
```

6.2 The Continuation Moving Protocol

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

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 components "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. 4 shows the connections between a mobile thread and its base.

Note that our continuation moving protocol expresses *computation mobility* in terms of "client-server role alternation", i.e. by specifying which end plays which of the roles at a given time. In principle, with some added complexity, our target-side `remote_call` operation can be replaced with equivalent *servant*-based target side code (see subsection 2.1) if security restrictions or unreachability of the target through firewalls gets in the way.

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

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

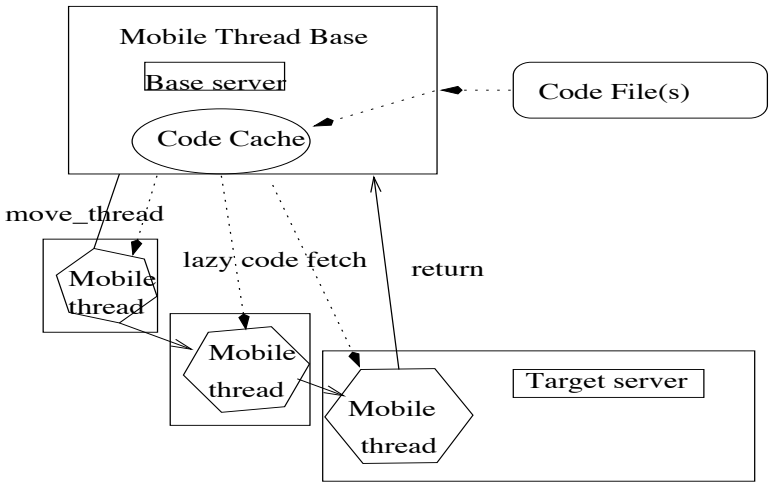

Fig. 4. Launching a mobile thread from its base

As our networking infrastructure, our *mobile threads* are platform independent. As Java, BinProlog is a platform independent emulator based language. As a consequence, a thread can start on a Unix machine and move transparently to a Windows NT system and back. Binaries for various Unix and Windows platforms are freely available at http://clement.info.umoncton.ca/BinProlog. For faster, platform specific execution, BinProlog provides compilation to C of static code using an original partial translation technique described in [24].

6.3 An Example

Networking code expressed in terms of mobile threads tends to be very compact. The following (self-explanatory) example illustrates both lazy code fetching and thread movement.

```
% assumes a server has been started on the same machine with:
%
% ?-port(9300)=>run_unrestricted_server.
%
% Program to be fetched over the network
% and run on target server
%
a(1).
a(2).
b(X):-a(X).
work_there:-b(X),write('X'=X),nl,fail.

% main code sending a moving thread to server
```

```
go:-
  % port where the target server runs
  port(9300)=>
    wrap_thread((     % wraps up code subject to movement
      write(here),nl, % action on 'base'
      move_thread,    % actual thread movement
      work_there      % action on target
  )),
  % code to be executed when 'back'
  write(back),nl.
% end of program

% Interaction on TARGET
?-port(9300)=>run_unrestricted_server.
.....
X = 1
X = 2

% Interaction on BASE
?-go.
here
running_server(port(9390),password(pw_5473)) % sets up base server
consulting(../progs/tmob.pl) % sets up code cache
consulted(../progs/tmob.pl)  % by consulting into a local database
time(consulting = 30,quick_compiling = 0,static_space = 0)
server_done(port(9390),password(pw_5473)) % password protected stop
back                                      % message back home
yes
```

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.

7 Mobile threads – are they needed?

Advanced *mobile object* and *mobile agents* agent systems when built on top of Java's impressive dynamic class loading and its new reflection and remote method invocation classes like 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?

7.1 The "pros" for *computation mobility*: simplicity and learnability

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.

7.2 Emulating computation mobility through control mobility

As shown in [18], 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*.

Overall, our belief is that availability of *computation mobility* can help both with simplifying programming and with making distributed applications more efficient and fault tolerant.

8 Mobile Agents

Mobile agents can be seen as a collection of synchronized *mobile threads* sharing common state [16]. We have first implemented them by using an emulation of *computation mobility* in terms of *control mobility* as described in subsection 7.2.

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.

Among the applications of mobile agents easy to express in our framework:

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

- enhancing the Web with virtual places
- avatar scripting in virtual worlds
- collecting data from very large databases distributed over the network through local interrogation
- network monitoring
- remote on-site assistance
- electronic markets
- knowledge discovery/data mining
- tele-teaching

9 Related work

A very large number of research projects have recently started on mobile computations/mobile agent programming. Among the pioneers, Kahn and Cerf's Knowbots [10] Among the most promising recent developments, Luca Cardelli's Oblique project at Digital and mobile agent applications [1] and IBM Japan's aglets [8]. We share their emphasis on going beyond *code mobility* and *control mobility* as present in Java and its RMI, for instance, towards *computation mobility*. Mobile code technologies are pioneered by General Magic's Telescript (see [9] for their last Java based *mobile agent* product). 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 achieve similar results 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 [18,17,19]) 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 [2]. 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 the formalism. Our *coordination logic* of [16] describes surprisingly similar ideas, 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.

10 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 OO languages having threads, reflection classes and remote method invocation. Future work will focus on intelligent mobile agents integrating knowledge and controlled natural language processing abilities, following our previous work described in [5,13,21].

Acknowledgment

We thank for support from NSERC (grants OGP0107411 and 611024), and from the FESR of the Université de Moncton.

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. L. Cardelli. Mobile ambients. Technical report, Digital, 1997. http://www.research.digital.com/ SRC/personal/Luca_Cardelli/Papers.html.
3. 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.
4. N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, 1989.
5. V. Dahl, A. Fall, S. Rochefort, and P. Tarau. A Hypothetical Reasoning Framework for NL Processing. In *Proc. 8th IEEE International Conference on Tools with Artificial Intelligence*, Toulouse, France, November 1996.
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. IBM. Aglets. http://www.trl.ibm.co.jp/aglets.
9. G. M. Inc. Odissey. 1997. available at http://www.genmagic.com/agents.
10. 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.
11. 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.

12. A. Mycroft and R. A. O'Keefe. A polimorphic type system for prolog. *Artificial Intelligence*, (23):295–307, 1984.
13. S. Rochefort, V. Dahl, and P. Tarau. Controlling Virtual Worlds through Extensible Natural Language. In *AAAI Symposium on NLP for the WWW*, Stanford University, CA, 1997.
14. 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*.
15. 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.
16. 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. to appear in LNCS, Springer.
17. P. Tarau, V. Dahl, and K. De Bosschere. A Logic Programming Infrastructure for Remote Execution, Mobile Code and Agents. In *Proceedings of IEEE WETICE'97*, Boston, MA, June 1997.
18. 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.
19. P. Tarau, V. Dahl, and K. De Bosschere. Remote Execution, Mobile Code and Agents in BinProlog. In *Electronic Proceedings of WWW6 Logic Programming Workshop, http://www.cs.vu.nl/ eliens/WWW6/papers.html*, Santa Clara, California, Mar. 1997.
20. 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.
21. 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.
22. 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.
23. P. Tarau and K. De Bosschere. Virtual World Brokerage with BinProlog and Netscape. In 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/ lp-net.
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.
```