In CICLOPS 2002 (ICLP Workshop), Copenhagen

A Simple Approach to Distributed Objects in Prolog*

Manuel Carro Manuel Hermenegildo

Computer Science School
Technical University of Madrid
Boadilla del Monte, E-28660,Spain
{mcarro,herme}@fi.upm.es

Abstract. We present the design of a distributed object system for Prolog, based on adding remote execution and distribution capabilities to a previously existing object system. Remote execution brings RPC into a Prolog system, and its semantics is easy to express in terms of well-known Prolog builtins. The final distributed object design features state mobility and user-transparent network behavior. We sketch an implementation which provides distributed garbage collection and some degree of tolerance to network failures. We provide a preliminary study of the overhead of the communication mechanism for some test cases.

Keywords: Objects, Distributed Execution, Remote Calls, Migration, Garbage Collection.

1 Introduction

Distributed objects are the natural sequel to object oriented programming and distributed programming. They can be used to implement a wide range of software systems: remote databases, service migration to improve access speed, automatic caching, implementation of stateful agents, etc. A good deal of proposals combining distribution and objects have been developed in the realm of procedural and OO languages; however, many of them boil down to adding convenient libraries to an existing host language which did not have any provision for distributed execution [BGL98]. Among the proposals which address OO and distribution from scratch we may cite Emerald [Jul88], Obliq [Car95], and, to some extent, Jini [We+00].

Although there have been several proposals for coupling LP and OO in the logic programming arena (e.g., SICStus objects [Swe99], Prolog++ [Mos94], LogTalk [Mou00], O'Ciao [PB02], and others), few proposal have been made, to the best of our knowledge, to couple objects and distribution. We can cite the mobility capabilities of Jinni [Tar99], and the distributed object system of Mozart Oz [RHB00,RHB+97]. The model we propose shares some points with Jinni (because it builds on Prolog syntax and semantics of the host language) and also with Mozart Oz (because of the way objects are perceived). It has, however, several characteristics which make it different from both. What we aim at is a simple scheme for objects, distribution, and mobility which does not depart too much from the semantics of Prolog, and which has some tolerance to external faults.

The techniques we will show are relatively easy to incorporate in a modern Prolog system, but we will occasionally resort to certain *Ciao Prolog* [HBC+99] features. *Ciao Prolog* subsumes ISO-Prolog, supports concurrency with communication through the (shared) built-in database [CH99], features an object system [PB02], attributed variables [Hol92], and a mature WWW/Internet interface [CHV99]. *Ciao Prolog* incorporates also a module system that allows defining syntactic (and semantic) extensions in a very user-friendly way [CH00]. In particular, modules can have local operators and translation rules which are used only at compile time and under the compiler control. These facilities are particularly helpful to define in a modular way higher level programming constructs. The language also allows assertions [PBH00] describing types, modes, determinacy, non-failure, execution cost, program documentation, etc. The assertions are also defined in source-level modules, and are thus extensible. Many of these characteristics facilitate the implementation of the model proposed.

The rest of this paper is organized as follows: Section 2 gives some background on the object system of *Ciao Prolog*; Section 3 describes the basic mechanism for communication with servers; Section 4 describes the design of the distribution primitives; finally, Section 5 shows preliminary performance measures.

* The authors were partially supported by MCYT projects TIC-99-1151 and HI2000-0043 (*EDIPIA* and *Adela*) and by ESPRIT project IST-2001-34717 (*AMOS*).

Primitive	Meaning
`:- class(stack)`	Starts the definition of the class *stack*
`X new stack`	*S* is a new object instantiating *stack*
`X:op(Args)`	*op(Args)* is executed on object *X*
`X:destroy`	*X* is destroyed (invoked explicitly or on GC)

Table 1. Builtins to handle objects

2 The *Ciao Prolog* Object System

As we will use the notation and semantics of the *Ciao Prolog* object system (O'Ciao) in Section 4, we will summarize the basic ideas behind it here (see [PB02] for more details). We will cover only class declarations, object creation, and object access. Many useful characteristics of O'Ciao will be left out: inheritance, virtual interfaces, user-defined object identifiers, etc., some of which need an involved implementation, including hierarchy trees and dispatching tables, runtime support, state initialization, etc.

The notion of object in O'Ciao is an extension of that of a module, its state being that of the dynamic facts in the module.[1] Dynamic facts not explicitly exported are local to the module where they are declared. Therefore, their scope allows maintaining a private state not accessible from outside unless explicitly stated so. Objects are constructed as *instances* of modules with a private state (i.e., a set of dynamic facts), which is reachable only through the module (i.e., class) interface. Different instances of modules are distinguishable by means of system-wide unique identifiers, and the state of every instance is private to that instance. There is a special syntax (Table 1) that allows defining classes, creating objects, and invoking methods (e.g., Prolog procedures) on them. Method visibility is controlled by simple scope rules: only exported predicates are callable from outside. Since objects are a grow-up of modules and dynamic facts, their behavior is easy to understand and to reconcile with that of traditional Prolog.

Let us examine an example: the code in Figure 1, left, defines the *class of card decks*, with two public methods. The code on the right creates two instances of the `deck` class (two decks of cards, accessible through variables `S1` and `S2`). A card is then drawn from one of the decks and placed in the other. This changes the state of both decks; these states do not interfere with each other, although they are defined using the same code. Note that objects can be created and used from code that is not an object itself, and vice versa.

The current implementation transforms the source code of a class in a user-transparent way. Relevant predicates are given an additional argument, which will hold at run time the (unique) identifier of the object, generated by the `new/2` primitive. This identifier is used to distinguish each object state from that of other objects of the same class.

```
:- class(deck,[addcard/1,drawcard/1]).

:- dynamic card/2.

card(1, hearts).     % initial state
card(8, diamonds).

addcard(card(X,Y)):-  asserta(card(X,Y)).
drawcard(card(X,Y)):- retract(card(X,Y)).
```

```
:- module(player, [main/0]).
:- use_package(objects).
:- use_class(deck).

main:-
    S1 new deck,
    S2 new deck,
    S1:drawcard(C),
    S2:addcard(C).
```

Fig. 1. A deck of cards implemented as an object (left) and code using it (right)

[1] In what follows we will be only interested in dynamic facts, although most of the discussion is applicable also to dynamic predicates.

3 Predicate Servers and Basic Communication

We want to achieve a communication scheme among distributed objects which gives network transparency to the user. In this section we will sketch how that communication can be performed. For simplicity we will not introduce objects yet, and we will use instead processes which *serve* predicates and processes which call these served predicates remotely. We will then present the basic communication mechanism and describe the server behavior without having to take into account complexities brought about by object management. Most of the ideas in this section can be lifted to the object-oriented case.

We assume the existence of a number of *servers*: processes that listen on a designated TCP/IP port, accept *queries* of *clients*, and react accordingly to them, either interpreting them directly or taking them as goals to be executed. A predicate server basically behaves as a traditional Prolog toplevel: wait for a query, execute it, return the answer to the caller, and wait again. A *served predicate* is a predicate whose code is accessible to the server and for which the server provides external access. Communication among caller and server can in principle use any sensible format, but an structured, easy to parse one is advisable in terms of speed (see Section 5).

Clients which invoke served predicates are farming out computations and using resources of the machine where the server lives. This also allows clients to access data pertaining to different hosts homogeneously across Internet by means of predicate servers. We will assume that a server is uniquely identified by an address (of the form, e.g., `host_name:port`) and that this address is known by all clients willing to access the server.

```
:- use_package(ciao_client).
:- use_module(library(system_info),
              [current_date/1]).

access(Site, Delta, LastVersion):-
   last_version(LastVersion) @ Site,
   current_date(D1) @ Site,
   current_date(D2),
   Delta is D2 - D1.
```

Fig. 2. Accessing remote predicates

```
:- module(sample_server,
         [last_version/2,
          current_date/1]).
:- use_package(ciao_server).
:- use_module(library(system_info),
              [current_date/1]).

last_version(1.1).
```

Fig. 3. Serving predicates

3.1 Calling Served Predicates

The syntactical construction we will use to express that a computation is to be performed at a server is the *placement* operator: `G @ Site` specifies that goal `G` is to be executed at the address `Site`. The caller is blocked until the answer is retrieved from the remote site; i.e., the call is synchronous. Declaratively `G @ Site` behaves similarly to `call(G)` for side-effect-free goals: bindings created with the execution of `G` at `Site` are seen locally upon end of the call, failure of `G` at `Site` leads to failure of `G @ Site`, and exceptions are also propagated locally. However, side effects (e.g., assertions or device access) caused by the execution of `G` take place at `Site` (which is needed in order to have the distributed object semantics shown in Section 4).

The definition (and operator declaration) of `@/2` are located in the libraries `ciao_server` and `ciao_client`. Figures 2 and 3 show code for a client and a remote server. The client code (on the left) accesses a server located at `Site`, calls a predicate (`last_version/1`) stored there, and computes the difference between the local and server dates. The code on the right corresponds to the server and exports the predicates called by the client.

There are several features which can be added to the remote communication scheme with very little cost. We will mention some of them because of their usefulness.

A Shorter Notation for Remote Modules It is easy (and maybe a common scenario) to state at compile time that a predicate always lives in a fixed remote site. All that is needed is to understand a declaration such as

```
:- use_remote_module(server_name, [pred/n])
```

which states that `pred/n` is located at `server_name` to translate a call to `pred(Arg)` into `pred(Arg) @ server_name:default_port`. This approach to remote execution was termed *active modules* [CH01,CB01].

Remote Execution and Concurrency Since `@/2` is blocking, but the computation is farmed out, local concurrency can be used to perform other tasks while data is on transit or execution is proceeding remotely, thereby reducing the impact of network latency and optimizing resource usage. As an introductory example, Figure 4 initializes two remote sensors at different sites and starts daemons to monitor them. (Remote) initializations are launched in separate local threads, and monitoring starts when the setup phase of both monitors has finished completely. The concurrency primitives we will use here have been proposed in [CH99,HCC95].

`G &&>` H_G and H_G `<&&` denote, respectively, task creation and data-driven task dependency. These operators are defined in a user-level library, and based on lower-level blocks. The first construct starts goal `G` in a new, independent thread while the local task proceeds, and leaves in H_G a *handle* to the goal `G`. Argument passing to the new thread is performed transparently to the user. H_G `<&&` waits for `G` to finish and (similarly to the `@/2` operator) makes results of the execution of `G` (bindings, failure, and exceptions) available.

```
system_monitor(Site1, Site2, TLimit):-
   setup_sensors @ Site1 &&> S1,
   setup_sensors @ Site2 &&> S2,
   S1 <&&, S2 < &&,
   watch_temp(TLimit) @ Site1 &&,
   watch_temp(TLimit) @ Site2 &&.
```

Fig. 4. Monitoring remote sensors

`G &&` behaves similarly to `G &&>` H_G, but no handle is returned, and therefore no further communication is possible. In Figure 4 remote calls are executed in separate threads (`B @ S &&>` $H_G \equiv$ `G &&>` H_G, therefore `G` $\equiv$ `B @ S`), the two remote initializations can be executed simultaneously, and the local thread proceeds when they finish. Handles make explicit a relationship among data dependencies and control, and can be used reorder (concurrent) goals so as to improve the amount of distribution / parallelism [Zur02].

Threaded Servers As a toplevel, a predicate server would block while executing a client goal. The standard solution to handle several incoming requests is to thread the server; however, this could lead to data conflicts both in the server code and in the data managed by the served predicates. Internet (e.g., ftp, WWW, ...) servers are explicitly designed to avoid these conflicts by using locks on shared variables or by forking out children in a separate address space. Several levels of interaction between served predicates can be identified. While this classification can be made of a finer grain, it will be enough to illustrate what information a predicate server would need to guarantee a safe execution.

We will resort to adding declarations to the main server module (the one in Figure 3) to specify which predicates can (not) be executed concurrently. The assertion mechanism [PBH00] of *Ciao Prolog* is a convenient means to do that: new declarations and their treatment can be defined in library modules. The declarations in the server code could be added either by the programmer or by an analysis tool, and be conveniently processed at compile time:

`:- pred quicksort/2 + reentrant.`
: Calls to `quicksort/2` can execute concurrently with any other calls at any time. The same would apply to other pure (non-side-effects) goals.

`:- pred update_my_db/2 + non_reentrant.`
: Calls to `update_my_db/2` are to be executed in mutual exclusion. This is a conservative, safe approximation for side-effect predicates which can potentially cause data conflicts.

On the other hand, served predicates that start threads on their own must ensure their correct behavior. Besides, a client could try to start a non-thread-safe predicate in a separate thread *in the server* with a call such as `G && @ Site` ($\equiv$ `(G &&) @ Site`). But, as `&&/1` is just a library predicate which cannot be called by a client unless exported by the server, disallowing these calls boils down to non importing the relevant module.

Sending and Receiving Terms with Attributes Goals to be remotely executed can be automatically inspected for the presence of attributed variables [Hol92], and the associated attributes (including attributes of variables present into attributes, and so on) can be sent and rebuilt upon reception. The ability to send and retrieve attributes is interesting because of their use to implement constraint solving capabilities, delay mechanisms, etc. It is possible, for example, to send and receive constrained terms just by sending them together with their attributes, which in many cases represent a completely self-contained projection of the constraint store over the variables in the original term.

3.2 Implementation Notes

The implementation of `G @ S` can be described with the following piece of code:

```
G @ S :- send_remote(goal(G), S, G'),
         ( G' = success(G) -> true ;
           G' = exception(E), throw(E)).
```

`send_remote/3` sends (a copy of) goal `G` as a term and waits for the returned answer. This answer can be `success/1`, meaning that the remote call returned with an exit substitution, `fail/0`, which will cause local backtracking, or `exception/1`, if the remote goal raised an exception uncaught in the server. The server is quite straightforward: it waits for goals, executes them, and returns answers. The overall procedure is depicted in Figure 5.

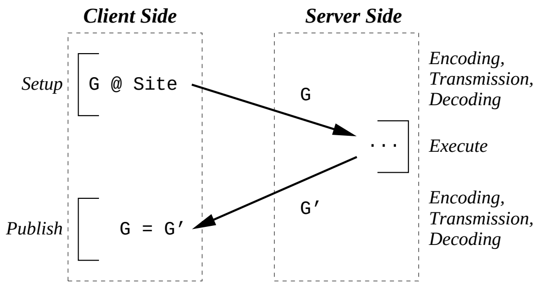

Fig. 5. How remote goal execution works

`send_remote/3` also takes care of extracting the attributes of the goal in a first-order term. Backtracking of remote goals has to take place in the server, which might have several pending, unfinished calls. This is best handled with multiengine support (which Ciao Prolog has, as it is needed to implement concurrency) which provides independent engine creation and backtracking.

Terms can obviously be sent and received using a standard ASCII-based representation with `write/2` and `read/2` (which we will call *plain representation*); however, the textual format is often relatively expensive in terms of time, due to e.g. tokenization and the overhead of having to cater for (perhaps inexistent) operators. A *marshaled* representation, which packs a term into a linear stream such that reconstructing the terms is easy (it does not have operators, commas, or parentheses, and each symbol is preceded with its type and arity) has proven to be comparable in terms of size and advantageous in processing time for other purposes (namely, to store compilation files intended to be read by the Ciao compiler). `send_remote/3` takes care of encoding/decoding terms using such a representation, which is basically the same as the one used by SICStus in the fast term I/O routines.

4 Objects and Distribution

Remote predicate calls provide additional computing power and resource access across the Internet. However they have some drawbacks: state migration to other servers is difficult to implement (the same module might already be present in the destination, for example), and they cannot maintain state separation but at the expense of contaminating the interface with per-client unique identifiers.

In this section we will devise a notation to include notions of distribution in the object framework of Section 2. We will assume a general infrastructure composed of (in general) several *object servers* where *objects* (state and method code) are actually held. This avoids creating per-object processes and makes objects lightweight. *Client* processes access servers where new objects can be created, invoke methods on objects, and move objects to other servers. All these interactions go through the corresponding *object server*, which takes care of message forwarding and garbage collection, when needed. We assume that the code of the class the objects instantiate is present in the server at compile time, and does not need to be transmitted

Primitive	Meaning
`X new Class @ Site`	Create an object *X*, instance of class *Class*, at *Site*
`X @ Site`	*X* is placed at *Site* (either returns *Site* or moves *X*)
`X send_ref Site`	Informs the server of *X* that we will send a reference to *Site*
`X meta_id Meta`	*Meta* includes the object identifier *X* plus its current location
`X provides Service`	register object *X* as providing *Service*
`X accesses Service @ Site`	which object *X* at *Site* provides *Service*?

Table 2. New object and distribution primitives

over Internet (as it is done in, e.g., Mozart Oz). Note that code transmission is not actually a technical problem, since Prolog systems can traditionally include easily interpreters or compilers in executables.

We pursue a final design that offers the programmer a level similar to that of non-distributed execution, thus facilitating the implementation of network-based applications. We will discuss some problematic issues and sketch solutions for them. For the sake of simplicity, we will ignore authentication, although very important in practice: we will assume that all clients have permissions to perform whatever operations are possible on object servers, and we will focus on how these operations can be performed.

4.1 A Notation for Remote Objects

Clients receive an identifier for every object they create. This identifier must be unique in the system and immutable, so that equality can be used to implements object identity. Clients communicate with objects with a set of primitives that are either new (Table 2) or which expand those for regular, local objects:

`Obj new Class @ Site` allows the `new/2` operation to create a new object at `Site`, where a server should be running. `Obj` is the object identifier, which is location-aware, i.e., it has additional information to locate where the object is actually stored. Uniqueness of identifiers can be ensured by using other (unique) identifiers such as host addresses, process number in a host, etc., to construct new identifiers.

`Obj:M` invokes method `M` on `Obj` and waits for an answer. The remote location does not have to be explicit (e.g., `Obj:M @ Site`) since it can be found through `Obj`. The distributed object library will locate the relevant server, communicate with it, and pass on the method call.

`Obj @ Site` expresses object placement, and it is used to move the object represented by `Obj` from its current location to `Site`. If `Site` is a free variable, it is bound to the name of the current location of the object. In the former case, the holder of `Obj` is contacted and a migration to the new server started; this takes place at the server level. It should be noted that the object state, being fundamentally a collection of facts, can be very easily transmitted over the Internet. Method calls on relocated objects are transparently forwarded to the appropriate location, and arguments are sent to and from the object, as in the remote predicate case (Section 3.1).

Destruction of remote objects by a client is performed by contacting the object server and issuing the necessary actions there. Object destruction can be triggered either by local garbage collection (GC) or by an explicit call to `destroy/1`. In general, a client's request of object deletion does not remove the object immediately, since other clients may be accessing the object. We will come back to this issue in Section 4.4.

Mobility in Other Systems Object relocation differs from that of Mozart Oz in that Mozart moves the object to the computation site of the caller thread [RHB+97,RBHC99] (i.e., object movement is implicit, while our proposal makes it explicit). This simplifies parts of the protocol and improves locality in many undoubtedly relevant cases [HRBS98]. However it also could turn out to be a drawback in some situations: e.g., centralized data accessed by different clients would bounce, generating additional traffic on the net, unless protected inside a *port*. Also, calls to objects which are part of a hierarchy would probably make the whole hierarchy migrate to the caller space.[2] If desired, automatic object migration can be expressed in our approach by means of an invocation operation `::/2` defined as

[2] There is an extension to the Oz mobile object protocol designed to achieve better performance in this case [HR99].

```
X::M :- X @ localhost, X:M.
```

which moves object `X` to the site executing the caller thread.

Jinni's mobility model actually moves the computation thread to and from a designated server. Coordination and data exchange among several threads can be performed thanks to Linda-like [CG89] primitives and an implementation of a shared blackboard.

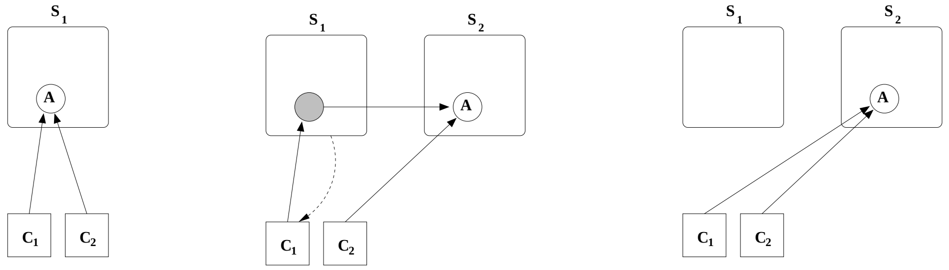

Fig. 6. Moving an object and forwarding requests

4.2 Object Migration

Migration takes place by means of a protocol between object servers which replicates the object state. In order to move an object, the server blocks new requests to the object and waits until the current call has finished before transferring the object state. Servers keep, for each object, a list of entities (external processes or other objects) that are known to have a reference to the object. This list (termed the *scion list*, in GC jargon) is initialized when the object is created, and is updated when an unlisted client accesses the object. It is used to ensure that objects are not deleted from the server when clients may still access them. When an object **A** is moved by a client **C**, the reference to **C** is also transferred to the new location. The former server of a relocated object can keep a *trace* **A'** of the original object together with part of the scion list (the initial list without the reference to **C**). Any client which has a reference to the trace is transparently notified of the relocation at the next connection attempt, and the corresponding entry of the scion list is transferred to the new location. The object trace is removed when its scion list is empty, because all the clients know about the migration.

Figure 6 shows an example. In the initial state, left, $\mathbf{S}_1$ holds object **A** which is referenced by $\mathbf{C}_1$ and $\mathbf{C}_2$. $\mathbf{C}_2$ moves **A** to $\mathbf{S}_2$. $\mathbf{S}_1$ acts as a proxy, keeping a trace of **A** (middle). When $\mathbf{S}_1$ receives a message addressed to **A**, further communication (dotted arrow) updates the view of $\mathbf{C}_1$ about the current placement of **A** (right) and the trace is deleted.

Unbound length chains caused by repeated object relocations can be avoided by making servers take into account where objects came from. In Figure 7, client $\mathbf{C}_3$ relocates an object from $\mathbf{S}_2$ to $\mathbf{S}_3$, and $\mathbf{S}_2$ informs $\mathbf{S}_1$ about the new location, thus shortening the reference chain (i.e., messages are forwarded to $\mathbf{S}_3$ by $\mathbf{S}_1$ without having to go through $\mathbf{S}_2$). Emerald [Jul88] applies a similar scheme.

Object addresses change after relocation, and since object identifiers must be immutable, addresses cannot be encoded within the identifier. A simple solution is to store the up-to-date object address as part of an internal table of tuples $\langle ObjId, ObjRef \rangle$ which is kept in each and consulted client by the remote object library as needed.

4.3 Communicating Object References

Clients in a cooperative environment should be allowed to pass object identifiers to other clients. Sending them in the presence of object mobility can cause races [AR98] that result in trying to access inexistent remote objects. Consider the scenario in Figure 8: $\mathbf{C}_1$ gives $\mathbf{C}_2$ the identifier plus remote location of **A**

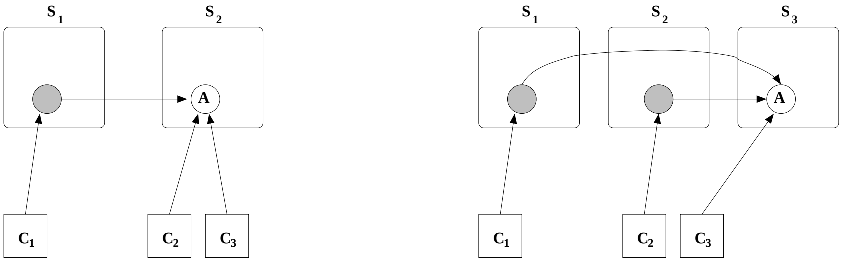

Fig. 7. Shortening forwarding chains

(denoted by **@A**); before $\mathbf{C}_2$ accesses **A** (so that $\mathbf{S}_1$ can register it in the scion list), $\mathbf{C}_1$ moves **A** to $\mathbf{S}_2$. $\mathbf{S}_1$ does not keep a trace of **A** (since the only known client with access to it has just requested a relocation), and $\mathbf{C}_2$ founds that neither **A** nor a trace is present.

Simple protocols to avoid this situation involve sending control messages both to the receiver of the object identifier and to the server [BGL98]. One additional primitive allows clients to hand out object identifiers to other clients with the cooperation of the server. `send_ref(Object, Client)` creates an entry in the scion list of `Object` pointing to `Client`. The situation in Figure 8 cannot happen: if $\mathbf{C}_1$ performs

```
..., A send_ref C2, A @ S2,...
```

$\mathbf{S}_1$ will not remove the trace of **A** because there is a new reference to it from $\mathbf{C}_2$. This handshaking avoids undesired object removal at the expense of a more involved communication of object identifiers.

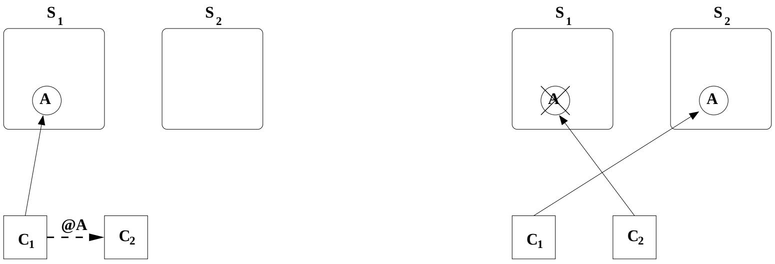

Fig. 8. Giving away a reference to an object which is suddenly moved

A client which receives the identifier of an object from another client does not have any means to access the object: object identifiers do not contain references to the actual object placement, as placement information is stored as a tuple in a hidden, library-managed table. It is necessary to send and *install* this tuple in the client which receives the object identifier. `Object meta_id AddressData` retrieves the placement data pertaining to `Object` (if `Object` is bound), or installs it locally and generates the corresponding `Object` identifier if `AddressData` is bound. It is intended to be used in conjunction with `send_ref/2` as in

```
communicate_object(Object, DestClient):-
   Object send_ref DestClient,        %% Tell server DestClient will know
   Object meta_id AddressData,        %% Retrieve placement data
   <...send AddressData to DestClient...>
```

and also by `DestClient` upon reception of the data representing the object placement.

4.4 Garbage Collection

Distributed GC [PS95] is more difficult than GC: in general the address space cannot be completely scanned. Additionally, distributed GC cannot be made both correctly and completely in the presence of link or node failures[Lyn97]. We will outline a simple method for automatic garbage collection based on the use of the scion list together with a timeout for the connections.

When a client destroys a remote object, its reference is removed from the scion list. If the scion list becomes empty, the object can clearly be removed (same as for an object trace). Each client listed in the scion list of an object is said to have a *ticket* on the object with an associated deadline. Renewing a ticket is performed by making any operation on the object. If a client does not renew a ticket on time, the server assumes the client has finished, crashed, or it is just disconnected, and the client reference is removed from the scion list. While this protocol can mistake a long enough transient network failure for a definitive broken link or client crash, real client crashes or permanent network disconnections will be treated properly.[3] It will also work in the cases where isolated objects reference each other (Figure 9): in that case a reference counting method or a scan-based algorithm will not work (because the reference count will never drop down to zero and because there is no bounded address space to scan). Timeouts will eventually remove all the references in the scion list, thus leading to object deletion. This management is similar to that used in the Java-based Jini architecture [We^{+}00] under the name of *leasing* (with the `Lease.ANY` time value). Starting with this idea, some points have to be addressed in order to make the system behave properly.

Keeping Objects Alive If some object is not accessed frequently enough by its clients as to renew the associated tickets, it could be (mistakenly) deleted once its deadline expires. Liveness is ensured by having a separate, library-managed thread in the client, which periodically traverses a (local) list of live objects and performs an empty action (e.g., `G @ _`) on every of them. Local GC, besides `destroy`ing the remote object should remove its identifier from that list — or just remove it and let ticket expiration remove the object. Should the client die unexpectedly, the thread would also die and the tickets expire, eventually leading to the object scion list shortening.

Using the Scion List to Ensure Liveness Objects referenced by other objects (as **B** in Figure 10) should also be taken into account for (non) deletion. While activity of client $\mathbf{C}_1$ can keep **A** alive, this does not necessarily mean that **B** will be contacted, and therefore could be removed if its deadline expires. The *liveness* of a client is to be transmitted to all the objects reachable from it.

Instead of broadcasting accesses from clients across all the referenced objects (redundant for multiply referenced objects), or starting a thread associated to every object (which could turn out to be too expensive), we propose using the scion lists, which maintain a graph expressing the relation "is referenced by". Every time a component of the scion list of an object **O** is candidate to be removed because its deadline has come, the graph that starts in that component is scanned (using, e.g., some variation of the *Probe and Echo* algorithm [Lyn97]) for live objects. If a live object is found by the scan, then **O** is potentially reachable, should not be deleted, and its ticket is renewed to the deadline of the found live object (in order to avoid objects with cyclic references to "pull" from each other). Note that this protocol takes place at the level of object servers, the objects themselves being unaware of it.

In Figure 10, **B** would face deletion if **A** does not access it frequently enough. However, a traversal following the dotted edges starting at **B** would find that **A** is alive, and therefore **B** should not be deleted. On the other hand, **C** will stay alive as long as **D** and **E** are. Eventually they will be removed, and so will **C**. The same idea can be applied at every step in the graph traversal so that several objects can be marked as live.

4.5 Name Registration

The scheme for method invocation in Section 4.1 cannot be applied when the clients are not designed to cooperate by, e.g., passing around object identifiers. This can be worked around by having the server to implement a *name service* that relates service names (which presumably do not change often) with the objects

[3] Other protocols would fail in the case of transient failures **and** in the case of definitive failures.

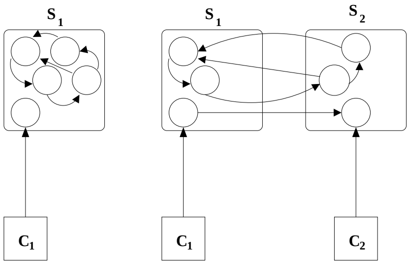

Fig. 9. Garbage cycles

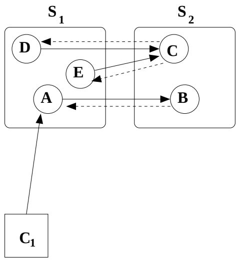

Fig. 10. Determining liveness. Dotted arrows are scion list components.

implementing the service. Once the object identifier is received, everything proceeds as before: the naming scheme just eases the (initial) access to an object. A client can start a remote object offering a service as:

```
:- use_class(share_market).

main:-
    X new share_market @ Site,   %% No outside visibility at the moment
    X provides gold_market.      %% A global name is assigned at Site
```

Clients which know a service name and want to access the provider object use the `accesses` operation to ask for an object reference. Servers keep a table of services with entries for objects and traces (thus allowing service forwarding). The entry is deleted when the object (resp. trace) is removed.

```
play_poker(PlayerSite, CardSite, Winner):-
    Deck new deck(poker) @ CardSite,
    Table new table(poker) @ CardSite,
    Deck provides deck(poker),
    Table provides rules(poker),
    P1 new player(Desk, Table) @ PlayerSite,
    P2 new player(Desk, Table) @ PlayerSite,
    P1:play(FinalScore1) &> P1H,
    P2:play(FinalScore2) &> P2H,
    watch(Table) &&,
    P1H <&, P2H <&,
    ( FinalScore1 > FinalScore2 -> Winner = P1 ; Winner = P2 ).
```

Fig. 11. Two players in a distributed game

4.6 An Example of Service Access

Figure 11 shows a piece of code that starts a deck and a table at some host `CardsSite`. Some players are also started as background threads, and the `Deck` object identifier is passed on so that they can draw cards. Interaction is made through the `Table`, which they can inspect and which gives turns to them. Both the `Deck` and the `Table` know which game to play from the initialization. The table is monitored by a local computation while the game is in progress.

5 Initial Performance Measurements

Let us remind that the basic remote call mechanism can also be used to implement state migration, and therefore it is important to assess its effectiveness and to fine tune it. Although communication delays make remote call more expensive than local argument passing, it is interesting to know precisely which overhead is to be expected (for, e.g., granularity control), and which points should be improved. We have measured the actual cost of performing remote calls with and without marshaling and attribute encoding (Section 3.1) in some benchmarks. Measuring the cost of attribute encoding is interesting, as attributes provide a means to create systems of distributed constrained objects, where constraints and constraint stores are represented with first-order terms. All the measurements were performed on a Pentium II at 333 Mhz with 64 Mb of RAM, running RedHat Linux 6.2 and using *Ciao Prolog* 1.7. The times were obtained by averaging ten runs.

The benchmarks send goals whose argument is a list (of different sizes), containing only integers (i.e., ground terms) or free variables (Table 3), and constrained variables (Table 4). In the third case, two different sets of constraints have been used: the first one constrains all the variables in the list to be greater than zero, and the second one constructs a list of variables that form a Fibonacci series with free initial values. The server does nothing with the received arguments: the served predicate is just a fact of the form `received(_)`. We measured the time needed to perform the full call: attribute encoding (when needed), marshaling (when done), sending the term, receiving the answer, and installing the bindings in the client. Both the server and the client were run in the same machine in order to minimize the impact of network latency. Running the same benchmarks in two comparable machines connected with a LAN gave practically identical results — in fact, slightly better.

Marshaling results in performance gain in all benchmarks but one (and the speed down is relatively small). This particular marshaling is more effective when many variables appear in the term. Its impact is therefore more important when no attributes are sent, since its encoding increases the ratio of ground (sub)terms. This is clear when constrained goals are sent, as their projection includes many constants (Table 4).

	Plain		**Marshaled**	
List	**No Attributes**	**Attributes**	**No Attributes**	**Attributes**
100 integers	10	18	12	14
500 integers	59	58	40	41
1000 integers	88	93	82	83
100 variables	34	52	12	23
500 variables	483	967	46	344
1000 variables	1508	3495	138	1234

Table 3. Performance for unconstrained terms (time in ms.)

List	**Attr., Plain**	**Attr., Marsh.**
10 vars. > 0	94	51
50 vars. > 0	672	548
100 vars. > 0	1996	1213
Fib (10)	261	219
Fib (50)	9153	8765
Fib (100)	43370	42112

Table 4. Performance for constrained terms (time in ms.)

List	**Proj. Size**	**Setup Time**	**Publish Time**
Fib (10)	3565	15	149
Fib (50)	24397	833	7531
Fib (100)	59176	2630	36597

Table 5. Projection size and time spent for different stages of the remote constraint service

The high cost of sending unneeded attributes in the case of terms with a large number of variables (Table 3) points to the necessity of using some form of control to decide whether to use attribute encoding or not. The overhead of sending constrained terms is remarkable, and we will examine more closely the cause of this relevant difference.

List	No Attributes	Attributes
100 integers	0	0
500 integers	0	3
1000 integers	1	8
100 variables	0	11
500 variables	0	508
1000 variables	1	2051

Table 6. Asssert-based transmission, no constraints

List	Attributes
10 vars. > 0	13
50 vars. > 0	307
100 vars. > 0	1164
Fib (10)	229
Fib (50)	12241
Fib (100	64198

Table 7. Assert-based transmission, with constraints

The size of the projections sent to and from the server could be blamed for the large delay. However, a measure of this size (Table 5) shows that there is no direct relation between it and the execution time. In fact, most of the time is spent in the local *Publish* phase (Figure 5), which in our case boils down to adding locally a set of equations that are already entailed by the store. The entailment tests take most of the time.

Additional experiments support the relative small cost of data transmission in comparison with constraint processing. We simulated remote calls with a definition of `@/2` which asserts and retracts terms (with attribute packing/unpacking) in the local database, instead of sending them through sockets: each send/receive pair is modeled as an assert/retract pair (Tables 6 and 7). We would expect this to be faster than network transmission, but in fact it is slower when attributes (which generate large terms) are taken into account (however, network transmission has a larger footprint when no attributes are sent). This places transmission cost in a fast network at a level roughly similar to that of database assertion (although in this case the database mechanism is probably doing more work than strictly necessary).

A solution to improve the cost of importing constraints through attributes, and which would also diminish speculative work in case of failure, is sending the constraints incrementally on both directions. Although techniques to instrument this transparently have already been studied [HCC95], the protocol becomes more involved (the interaction client/server would not finish when the goal returns, but it would continue by incrementally sending constraints until the goal finitely fails). Also, in the presence of backtracking some form of caching would be needed to avoid excessive communication overhead. An intermediate solution is to send the constraints as a whole to their destination, and publish locally the bindings on demand. This would avoid traffic of small messages, add constraints incrementally, and simplify parts of a completely incremental implementation.

6 Conclusions and Future Work

We have shown a simple approach to explicit distributed computation in Prolog based on providing predicate servers (for stateless entities) and distributed objects, featuring mobility, garbage collection, and tolerance to some classes of network and system faults. Our proposal tries not to depart too much from sequential semantics, and lets the programmer have explicit control about the concurrency and placement of computations. We feel that it is, however, of a high enough level so that most of the burden of concurrent/distributed programming does not fall on the programmer shoulders.

A preliminary evaluation of the performance of the communication facilities has been made, but a more complete and thorough study is to be carried out. Among the points to improve, a better, on-demand installation of variables and constraints is needed. Automatic code distribution, present in other systems, is to be also incorporated. While the fundamental pieces are already in the *Ciao Prolog* system (modules, analysis of interfaces, lazy loading, network access...), and can be used straight away, some issues of practical importance, such as authentication and code certification, have to be incorporated.

We wish to thank Francisco Bueno for fruitful (and never-ending) discussions regarding distribution and access to remote predicates, and the referees for their very accurate and detailed comments.

References

[AR98] S. E. Abdullahi and G. A. Ringwood. Garbage Collecting the Internet: A Survey of Distributed Garbage Collection. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[BGL98] J.P. Briot, R. Guerraoui, and K.P. Lohr. Concurrency and Distribution in Object Oriented Programming. *ACM Computing Surveys*, 30(3):330–373, September 1998.

[Car95] L. Cardelli. A language with distributed scope. *ACM Transactions on Computer Systems*, 8(1):27–59, January 1995. Also in POPL 1995.

[CB01] J. Correas and F. Bueno. A configuration framework to develop and deploy distributed logic applications. In *ICLP01 Colloquium on Implementation of Constraint and LOgic Programming Systems*, Ciprus, November 2001.

[CG89] N. Carreiro and D. Gelernter. How to Write Parallel Programs – A Guide to the Perplexed. *ACM Computing Surveys*, September 1989.

[CH99] M. Carro and M. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *1999 International Conference on Logic Programming*, pages 320–334. MIT Press, Cambridge, MA, USA, November 1999.

[CH00] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[CH01] D. Cabeza and M. Hermenegildo. Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. *Theory and Practice of Logic Programming*, 1(3):251–282, May 2001.

[CHV99] D. Cabeza, M. Hermenegildo, and S. Varma. The PiLLoW/Ciao Library for INTERNET/WWW Programming using Computational Logic Systems, May 1999. See http://www.clip.dia.fi.upm.es/Software/pillow/pillow.html.

[HBC+99] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[HCC95] M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. In *ICLP'95*. MIT Press, June 1995.

[Hol92] C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *1992 International Symposium on Programming Language Implementation and Logic Programming*, pages 260–268. LNCS631, Springer Verlag, August 1992.

[HR99] M. Hadim and P. Van Roy. A New Mobile State Protocol for Distributed Oz. In *PDCS 99*. August 1999.

[HRBS98] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3), May 1998.

[Jul88] E. Jul. *Object Mobility in a Distributed Object-Oriented System*. PhD thesis, University of Washington, 1988.

[Lyn97] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kauffmann, 1997.

[Mos94] C. Moss. *Prolog++*. Addison Wesley, 1994.

[Mou00] Paulo Moura. Logtalk 2.6 documentation. Technical report, University of Beira Interior, 2000.

[PB02] A. Pineda and F. Bueno. The O'Ciao Approach to Object Oriented Logic Programming. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*, Copenhagen, July 2002.

[PBH00] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[PS95] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In *International Workshop on Memory Management*, September 1995.

[RBHC99] P. Van Roy, P. Brand, S. Haridi, and R. Collet. A Lightweight Reliable Object Migration Protocol. In H.E. Bal, B. Belkhouche, and L. L. Cardelli, editors, *ICCL'98 Workshop on Internet Programming Languages*, volume 1686 of *LNCS*. Springer Verlag, October 1999.

[RHB+97] P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile objects in distributed oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.

[RHB00] P. Van Roy, S. Haridi, and P. Brand. *Distributed Programming in Mozart - A Tutorial Introduction*. DFKI, SICS, and others, February 2000. Available from `http://www.mozart-oz.org`.

[Swe99] Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog 3.8 User's Manual*, 3.8 edition, October 1999. Available from `http://www.sics.se/sicstus/`.

[Tar99] Paul Tarau. Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *PAAM'9*. The Practical Applications Company, 1999.

[We+00] Jim Waldo, Ken Arnold (editor), et al. *The Jini(TM) Specifications*. Addison-Wesley, December 2000.

[Zur02] Alberto Díez Zurdo. Goal Reordering in Logic Programs with Unrestricted Parallelism and Explicit Dependencies. Master's thesis, School of Computer Science, Technical University of Madrid, September 2002. In Spanish — In preparation.