Design of a Distributed Implementation of ABCL/1

Jean-Pierre BRIOT Jean de RATULD

Equipe Mixte LITP - RankXeroxFrance,
Université Paris VI,
4 place Jussieu, 75005 Paris, France
litp!{briot,ratuld}@inria.inria.fr.uucp

Abstract

In this paper we describe the design decisions leading to a distributed implementation of the ABCL language. We shortly review the ABCL (Actor Based Concurrent Language) computation model. We discuss some slight changes we bring to the current ABCL/1 language in order to distribute objects among several machines. Identification, reference, allocation and communication are the four concepts which need some evolution. The status of a first prototype implementation on a network of workstations (Lisp/Suns+Ethernet) is reported.

1 The ABCL Model of Computation

The ABCL/1 language is an object-based concurrent programming language (OBCP), based on actors [Agha 86]. Its main characteristics as an OBCP language are:

- **three types of message passing:**
  asynchronous (called *past* type), synchronous (called *now* type), and eager (called *future* type). The first type may include the specification of the *reply destination* of the message, i.e. to which object the result should be returned. The two latter types express an implicit reply destination,
- **two modes of message passing:**
  ordinary and *express*. The latter mode, a kind of interruption, allows monitoring and control of objects during any mode of activity.
- **hybrid computation:**
  more traditional procedural or functional style (usually Lisp) may coexist with objects and message passing to express the behavior of an object. Size and lifetime of objects is greater than in the Actor model of computation [Agha 86].

A complete description of the ABCL/1 computation model and programming language may be found in [Yonezawa et al. 86].

1.1 Defining Objects

The definition of an object in ABCL/1 is an expression `[object ...]`. The name (`counter`) is optional. Following is an example of a definition of a counter object:

```
[object counter
   (state [n := 0])
   (script (=> [:incr] [n := (+ n 1)]))]
```

1.2 Referencing Objects

The evaluation of the previous expression returns a newly created object as value. This object may be bound to a state variable declared in another object. If the definition appears at top-level and a name is defined, the object will be implicitly bound to some top-level global (and dynamic) variable with this name. The internal structure of the object (e.g. the instance of a structure, a cons, a vector, a closure..., depending on the implementation, currently in Lisp) is returned as the value of the definition expression. Thus there is a direct reference to the memory location of the object.

2 Distribution Issues

ABCL/1 was designed as an object-based concurrent programming language. However we need to bring some level of indirection in order to model remote references to objects in a distributed implementation.

2.1 Identification

In a distributed architecture (network of nodes), an object may reference another object which is located on some remote node. Consequently we need to be able to identify an object whatever is its location. The technique is to associate identifiers to objects. One or more tables, (called *object table(s)*), keep correspondence between identifiers and real locations of objects. Identifiers may be local or global:

global
: Some unique identifier is assigned to any newly created object. There is only one global object table.

local
: Each node owns a local object table. The identification of an object is a couple *index@node-name*, also called an *address*, where *index* is the entry in the object table of the node.

A global object table unifies manipulations of references but is a bottleneck in a distributed architecture. For every message transmission or object creation, the system must access to this global table.

The advantages of local tables are the simplicity and efficiency of remote access. All information is contained in the address itself. On the other hand because of the distribution of addresses, this complicates the search for references to an object for garbage collection and migration, as opposed to a simple update in a global table.

We chose solution of local tables. Now we got two more alternatives. Do we introduce addresses for every object or only for remote references?

The first choice main advantage is uniformity. The second choice decreases size of the object tables. Objects which are not remotely referenced may be garbaged by the local underlying Lisp garbage collector. This eases the design of the distributed garbage collector. On the other hand, this complicates the communication system which must transform references and possibly create new entries in the object table when passing local objects as parameters. There is a tradeoff about the amount of remote references to objects. If there is a high distribution of objects, the first solution may be better. We designed both alternatives, but the second one is our current choice.

2.2 References

When creating an object at top-level with a name, this implicitly binds this object to a top-level variable, local to the node of definition. Every node may be used as a top-level. Each top-level may load files and define objects, and multiple users may concurrently define objects from several top-levels. For each top-level node, a *top-level table* keeps the set of references from top-level. This defines the interface between the user and the objects to which he may send messages. It may be viewed as the environment (state variables) of an object which represents the user on the node. This top-level environment is also implicitly accessible by every object located on the node. This table will be used by the distributed garbage collector.

The user may also decide to explicitly reference some remote object bound at some remote top-level. The primitive `(remote-reference <top-level-variable> <node>)` allows him to do so.

2.3 Allocation

2.3.1 Creation

There are some important issues about creating objects. The system should be able to create objects on remote nodes (other than the node for definition). This could be implicitly managed by the system in order to have a good memory balancing at creation time. We believe however that the user should be able to explicitly specify on which nodes he wishes to distribute the components of the initial configuration of its application.

We give the user the ability to explicitly declare on which node the object will be created. This optional form is: `(node` *node-name*`)`. The user may disallow any remote creation by explicitly stating the allocation on the current node by declaring: `(node here)`.

```
[object counter
   (node sun4) (state [n := 0]) ... ]
```

Some default algorithm creates the object on the definition node but other algorithms are available to automatically improve memory-balancing. Intelligent algorithms should optimize both memory-balancing and load-balancing of processors, as well as minimize communications between objects by grouping communicating objects on nodes.

2.3.2 Migration

Memory/load-balancing predicted statically or defined at creation time may be not sufficient. Some more general load-balancing algorithm may decide to dynamically migrate objects among nodes. In case of a global object table, there is just need to update the entry in the global table. In case of local tables (our current choice), we would need to update all remote addresses (references) of this object. This cannot be done efficiently in practice. Therefore we must introduce forwarders, but this slows down communication. Migration of objects is not supported in our current prototype. Research on dynamic load-balancing algorithms is still an open issue. We believe that there could be interesting alternatives based on auto-regulation of dynamic systems. Note that ABCL/1 was designed with some higher granularity than actors in order to ease matching of objects onto processors and to minimize migration.

2.3.3 Disallocation

We designed a first prototype distributed garbage collector for our architecture. This is an extension of the usual mark and sweep algorithm for list processing languages. Our philosophy is to preserve the underlying Lisp garbage collectors on each node without any modification. We will add a distributed garbage collector for the local object tables. Its goal is to free the entries which are no more referenced in the object tables.

The distributed garbage collector algorithm is a mark and sweep. The mark phase consists in marking every entry (in object tables) which are referenced through the top-level. Every top-level will scan its top-level table, and for each address (of an object) referenced at top level, the entry in the object table corresponding to the address will be marked, and every referenced object will also be recursively scanned. The sweep phase starts when the marking phase is completed. Then all object tables are scanned and the entries which were not marked are freed. Such entries become free, i.e. ready to reference a newly created object. The corresponding objects become potential garbage for the Lisp garbage collector. Of course every entry in the object table is also unmarked to prepare the next marking phase.

Our current prototype is synchronous (all objects tables will be synchronously garbage collected). Completion of the marking phase is clearly defined. We are now studying current proposals of distributed garbage collection algorithms, based on copying, reference counting, and incrementality, in order to improve our current prototype.

2.4 Communication

The communication system is a *client-server* model using the Un*x sockets. The current implementation uses two Un*x processes per node. The *client* process is in charge of sending messages to remote hosts. The *server* process is waiting for the communication requests. Connection is established between clients and servers with Un*x sockets. Files are used as interfaces on each node between ABCL, the client and the server. A direct and more efficient interface is currently being implemented.

3 Conclusion and Further Work

This model of distributed ABCL was initiated with a micro-interpreter, called micro-ABCL, developed in the team of Pr. Yonezawa in 1986. The current prototype is implemented on a network (Ethernet) of workstations (Suns) in Lisp. We intend to start an implementation on multiprocessors after some evaluation of our current design.

We thank Alain Deutsch for his help on Un*x communication protocols.

References

[Agha 86] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, *Series in Artificial Intelligence*, MIT Press, 1986.

[Yonezawa et al. 86] A. Yonezawa, J-P. Briot and E. Shibayama, Object-Oriented Concurrent Programming in ABCL/1, *OOPSLA'86*, Special Issue of SIGPLAN Notices, Vol. 21, No 11, pp. 258-268, November 1986.