An Algorithm for Dynamic Reconfiguration of Mobile Agents

Marco Tulio Valente[1,2], Roberto Bigonha[1] and Mariza Bigonha[1]

[1]Department of Computer Science, Federal University of Minas Gerais
[2]Institute of Informatics, Catholic University of Minas Gerais
Belo Horizonte - MG - Brazil
E-mail: {mtov,bigonha,mariza}@dcc.ufmg.br

Abstract

In this paper we show an algorithm for dynamic reconfiguration of distributed applications based on the mobile agent model. We also show that the proposed algorithm can be easily implemented in the IPL language, a language with several abstractions for the construction of mobile applications in the Internet.

Key words: dynamic reconfiguration, mobile agents, Internet programming languages.

1 Introduction

Most distributed systems are designed to run without interruptions. In general, this kind of application can not be stopped even to apply corrective or evolutionary changes. As an example of such applications we can mention industrial process control systems, network management software, telecommunication switches etc. Thus it is important to be able to extend or modify these long-running systems without having to stop them. This capability is known as dynamic reconfiguration [Hof93].

On the other hand, recently mobile agents were proposed as an alternative model to design distributed applications in the Internet. A mobile agent is a process that can autonomously migrate among the nodes of the Internet to execute a task on behalf of its user [Whi97]. It is claimed that mobile agents can be used to build distributed systems that can reduce the network load, that are more robust to bandwidth fluctuations and that can also operate in disconnected mode. Among the applications that benefit from this new paradigm of distributed programming, we can mention monitoring and notification systems [LO99], i.e., systems where dynamic reconfiguration is an important requirement.

Traditionally, most of the work done in dynamic reconfiguration has focused on systems designed following the traditional client/server model. In this paper we present an algorithm for dynamic reconfiguration of distributed applications designed accordingly to the mobile agent paradigm. We also show that this algorithm can be easily implemented in the IPL language [VBLB00]. IPL is an object based language that has several abstractions to the construction of mobile applications in the Internet.

The remaining of this paper is organized as follows. Section 2 briefly presents the IPL language, which is used to encode the examples and algorithms presented in the following sections. Section 3 discusses the three kinds of dynamic reconfiguration possible in distributed applications. In Section 4, we present the algorithm for dynamic reconfiguration of mobile agents proposed in this paper. Section 5 describes the IPL implementation of the proposed algoritm. Section 6 reviews related work and Section 7 concludes the paper.

2 The IPL Language

IPL (Internet Programming Language) [VBLB00] is a language designed to construct mobile systems in the Internet, including applications in the mobile agent paradigm. The abstractions for mobile computation available in IPL resemble the style of computation proposed by the Ambient Calculus [CG98]. IPL is an object based language whose syntax follows Obliq [Car95]. Unlike Obliq, however, the language does not include abstractions that do not scale up to the whole Internet, like the notions of distributed lexical scope and network references.

Similar to Obliq, programs in IPL are organized as a set of objects. Since it is an object based language, IPL does not provide support to classes, inheritance or dynamic method dispatching. An object with fields $x_1, x_2, ..., x_n$ has the form:

{ x_1 `=>` a_1`,` x_2 `=>` a_2`,` ... `,` x_n `=>` a_n }

where each a_i can be a value or method field. A value field is defined like in the following example:

```
x => 3
```

A method field is defined in the following way:

`x =>` **meth** ($y, y_1, y_2, ..., y_m$) `b` **end**

where the parameter y is the *self* object, $y_1, y_2, ..., y_m$ are the remaining parameters and b is the method's body.

The main abstraction of IPL to support mobility is the notion of container. A container is a wrapper of objects that makes them mobile, that is, containers can move to *contexts* located in other network nodes. Contexts are processes that can receive, execute and send containers to other contexts, as showed in Figure 1. Contexts also offer *resources* to the execution of containers, like, for example, a window system or a data structure. Contexts are identified by URLs in the form: *host/name*, where *host* is the name of the workstation where the context is running and *name* is the name of the context.

A container with objects $p_1, p_2, \dots, p_n$ is created by the following command:

new_container $(m, p_1, p_2, \dots, p_n)$,

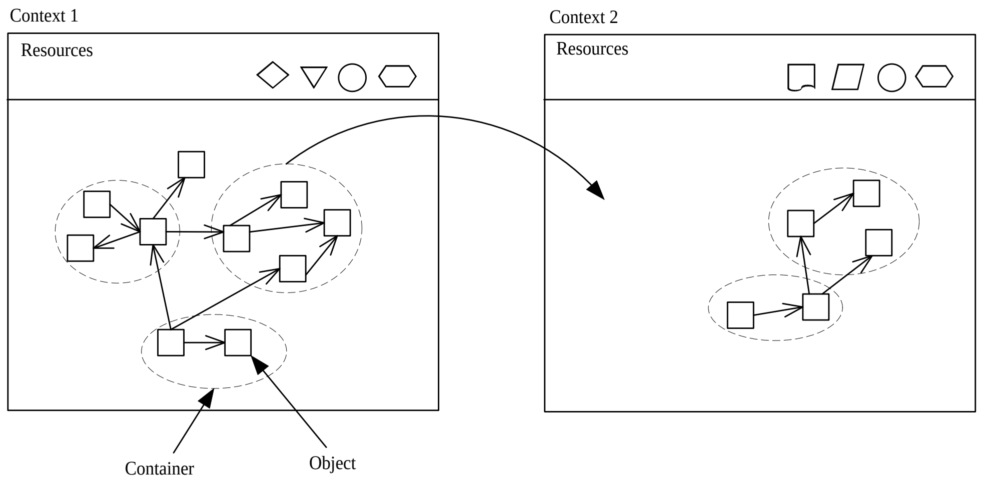

Figure 1: Abstractions for mobile computation available in IPL

where m is an optional parameter that denotes the name of the container. Container mobility is implemented by the following operations:

- `insert_object (c, a)`: insert the object a in the container c.
- `context_jump (d)`: moves the current container to the context d. After this operation, the execution continues in the next instruction, but in the target context d.
- `move (c, d, p)`: moves container c to context d. The execution in the current context continues asynchronously in the next instruction. In the target context, the execution begins by the the method *start* of the object p and finishes when this method returns.
- `this_container`: returns the name of the current container.

Objects in IPL are handled by *nominal semantics* [Gor00]. Every object has an implicit name that uniquely identifies it in any context of the network. A definition of the form $x = \{\ldots\}$ associates with variable x the name of the created object. When an operation of the form $x.op$ is executed, it is initially verified if an object with the name denoted by x exists in the local context. If such object exists, the operation op of this object is executed. In case that it does not exist, the call remains *blocked* until this object become locally available. Therefore, in IPL objects located in the current context are called *available objects*, while remote objects are called *unavailable objects*.

This semantics for object manipulation does not entail action-at-a-distance [Car99], i.e., it does not allow transparent manipulation of network references as usual in distributed languages for local area networks. Moreover, nominal semantics allows free migration of objects wrapped in containers, as it does not create “static links” among these objects and their execution context.

3 Dynamic Reconfiguration

There are three possible kinds of dynamic reconfiguration in distributed systems [Hof93]:

- Module replacement: when one or more modules of the application need to be replaced. For example, programmers may wish to replace a module by another one that implements a more efficient algorithm.
- Structural change: when the logical structure or the topology of the system may change. For example, new modules may be introduced and current modules may be removed.
- Geometrical change: when the mapping from the logical structure of the system to its distributed architecture may change. Usually, this form of reconfiguration is useful for load balancing, fault tolerance or to allow better use of communication resources.

To illustrate these different kinds of reconfiguration, suppose a distributed implementation for the dining philosophers problem [KM90]. In this problem, module replacement reconfigurations are needed when we decide to change one of the philosophers by another one with, for example, a better appetite. A structural change is needed, for example, when we decide do add or remove a philosopher from the system. Lastly, a geometrical change is required when we decide to move a philosopher from one node of the network to another one.

The last two kinds of reconfiguration – structural and geometrical – are already available in any mobile agent system implemented in IPL. Structural changes are supported by the notion of dynamic linking of containers, which allows containers to be added or removed from systems during execution time. Furthermore, the capability to move containers dynamically and autonomously from one execution context to another provides support to geometrical changes. For this reason, in this paper we focus only in reconfigurations that require module replacement.

The main problem to support dynamic module replacement in a system is related to transferring the execution state from one container to another. Particularly, this transfer should leave the new container in a consistent state, i.e., in a state where the execution of the system can proceed normally rather than going to a error state [KM90].

To illustrate the problems involved in transferring the execution state from one container to another, suppose a container c denoting a consumer agent:

```
1:  let p= { cont => 0,
2:           start => { resource buf;
3:                      var s;
4:                      while (true) do
5:                        s:= buf.get();
6:                        cont:= cont + 1;
7:                        "algorithm to process s"
8:                      end;
```

```
                 }
       }

let c= new_container (p);
```

A dynamic reconfiguration of this container may require replacing object p by another one with a different algorithm to process the value s removed from the buffer. Dynamic reconfiguration algorithms aiming to support module replacement should then provide answers to the following questions:

- How the execution state of a container is defined ? Certainly, this state should include the fields of the objects inside the container. But how to deal with fields in the old container that were removed in the new version ? And about fields whose types changed in the new container ? Considering these questions, a consensual decision is that the state information transfered to the new container should not include the program counter, since this value in the code of the new container can be associated to a completely different instruction.

- When the state of the old container should be transfered to its new version ? In order to avoid inconsistencies, one possible solution is to wait until the container becomes inactive, i.e., until no threads are running in the methods of the container. However, as shown by the previous consumer agent example, it is common to have infinite loops in notification and monitoring systems, making the containers of such systems continuously active. On the other hand, a reconfiguration can not happen at any state of the execution of the old container, since this can result in several kinds of inconsistencies. For example, supposing that the program counter is not transfered to the new container, a inconsistency happens in the previous example if the execution changes to the new container before processing an item removed from the buffer. In this case, this item will be lost during the reconfiguration.

Previous works about dynamic reconfiguration of distributed systems have already concluded that the problems mentioned above could only be solved if the participating containers provide information to guide the reconfiguration process [Hof93]. The old container should, for example, announce the moment when its state is consistent and therefore can be safely transfered to the new container. This state is usually called a *reconfigurable state*. The new container should then be responsible to access the state of the old container in order to copy all the information required to proceed the normal execution of the system.

4 The Proposed Algorithm

Suppose that we want to dynamically replace container n, containing objects $p_1, p_2, \ldots, p_m$ by another version with objects $q_1, q_2, \ldots, q_m$. The container is currently running in context t. In order to create the new version of the container, the following operation must be used:

```
let c'= new_container_config (n,q1,q2,...,qm);
```

After this first step, we should send the new configuration c' to context t using the *move* operation available in the language. In order to create a new container configuration, the programmer of the application should know the name n of the container and its current execution context. In this way, for security reasons, containers names should not be shared with any application.

As mentioned in Section 3 the old version of container n should explicit the states of its execution when a reconfiguration can take place. This is done by calling the function *reconfig_event()*. This function first verifies if there is locally a new configuration for the current container. If this is the case, the reconfiguration process is started. Initially, all messages in transit to container n are suspended until the end of the reconfiguration, when they will be delivered to the new version of the container.

Next, the *upgrade* method of each object $q_1, q_2, \ldots, q_m$ is called. These methods are executed following the order that the objects were inserted in the container, i.e., beginning by q_1 and ending by q_m. The *upgrade* method of each object q_i can use the identifier *old* to access the fields of the object that it is replacing. In this way, the *upgrade* method is used to transfer state information from the old version of the container to its new version. In the upgrade method it is also possible to access any field of object p_i in the old configuration by indexing the identifier *old* in the following way: *old* $[i]$. This is useful when the new configuration removes a object from the previous configuration, but requires its state to be transfered to one of the new objects. Once the execution of the *upgrade* methods are accomplished, the execution is resumed by the *start* method of the new configuration. The objects of the old container can then be garbage collected.

We show next an example of dynamic reconfiguration of the consumer agent described in Section 3:

```
1:  let q= { cont => 0,
2:           start => { resource buf;
3:                      var s;
4:                      while (true) do
5:                        s:= buf.get();
6:                        cont:= cont + 1;
7:                        "new algorithm to process s"
8:                        reconfig_event ();
9:                      end;
10:                    }
11:
12:          upgrade => { cont:= old.cont; }
13:        }
14:
15: let c'= new_container_config (c, q);
16: move (c', t, p);
```

Besides a new algorithm to process an item s (line 7), the code of the object q is instrumented with a call to the *reconfig_event()* function (line 8). This means that any further reconfiguration of this container will only happen after processing an item read from the buffer. The object q also includes an upgrade method that transfers the counter of items from the old to the new configuration (line 12). In this example, we suppose this is the only information that should be preserved by the reconfiguration process. It should also be noticed that the name c of the old configuration is needed to create its new configuration (line 15).

5 Implementation

The dynamic reconfiguration algorithm proposed in this paper benefits from the nominal and blocking semantics used by IPL in method calls. Since in IPL every object has a unique name in the network, we can easily block messages to an object by renaming it. In IPL, object and container names are keys with 144 bits organized in the following fields:

- bits 0 to 127: this first field is named OID (*Object Identififier*) and it uniquely identifies the object or the container in any node of the network. The value of this field is created using the algorithm proposed by the OSF DCE standard to generate GUIDs (*Globally Unique Identifiers*) [Ope97].
- bit 128: this second field is named *unavailability bit*. In objects, its main use is to disable the processing of messages. In containers, it is used to indicate that the container is a new configuration that is not enabled yet.
- bits 129 to 143: this third field is named CID (*Configuration Identifier*) and it stores a value that identifies the version of the container. This field does not have any meaning in object names.

The operation *new_container_config* creates a new container name from the current name. The new name has the same OID from the old name, but its CID is incremented by one. The unavailability bit of this new name is also set to one to indicate that the container is still unavailable.

The following auxilary functions are used in implementation of the reconfiguration algorithm:

- `_new_config (n)`: checks if there is in the current context a new configuration for container n, i.e., if there is locally a container with the same OID, but with a greater CID. If such container exists, the function returns its name; otherwise it returns zero.
- `_available (n)`: makes container n available by unsetting bit 128 of its name.
- `_unavailable (n)`: makes container n unavailable by setting bit 128 of its name.
- `_rename (p, q)`: renames the OID of object p to the same OID of object q.
- `_start_object (n)`: returns the name of the object with the start method of container n.

Using the previous functions, the implementation of the reconfiguration algorithm is straightforward:

```
1: proc OnReconfig(n) {
2:  n'= _new_config (n)
3:  if (n' > 0)
4:     _unavailable (n);
5:     for each pi in n,  _unavailable(pi);
6:     for each qi in n', qi.upgrade();
7:     for each qi in n', _rename(qi, pi);
8:     _available (n');
9:     _start_object(n').start();
10:}
```

The algorithm initially verifies if there is locally a new version for the container n (line 2). If such version exists, its OID is greater than zero and the reconfiguration process starts (lines 4-9). First, the current container and its objects are made unavailable (lines 4-5). Next, we execute the upgrade methods of each object of the new configuration (line 6). The objects of the new configuration are then renamed to the same name of the current objects (line 7). In this way, the new objects are going to process all suspended messages to the old objects and also the new messages that were sent to the container during the execution of the *OnReconfig* function. Last, the new container is enabled (line 8) and its execution starts by the start method (line 9).

6 Related Work

The current paper is inspired in a proposal to add dynamic reconfiguration in the distributed programming system Polylith [Hof93, Pur94]. Applications in Polylith are organized in a set of modules interconnected using a message bus. In the system, each module represents a process. Polylith supports the three kinds of dynamic reconfiguration mentioned in Section 3: module replacement, structural and geometrical change. Similar to the algorithm proposed in this paper, dynamic reconfiguration in Polylith can only happen in pre-defined states and it is not transparent to application programmers. Unlike dynamic reconfiguration in IPL, in Polylith the reconfiguration process does not start by merely plugging a new module in the system. Besides the code of the new module, programmers need to specify a script to guide the reconfiguration process. This script should be responsible for the following tasks: blocking communication among modules during the reconfiguration, rerouting messages to the new module that were in transit when the reconfiguration started and transferring the execution from the old to the new module.

Since the proposal presented in this paper rely on a distributed language designed with dynamic reconfiguration in mind, it does not require the use of such external script. The main reason is that the nominal semantics used to call methods in IPL makes it easy to suspend messages to the new container during the reconfiguration process. Also in IPL the transfer of the execution state to the new module is done by the upgrade method, i.e., by a internal method of the new version of the application and not by an external script.

Argus [BD93] is a system that supports dynamic reconfiguration of distributed applications in the CLU language. In Argus, it is possible to dynamically replace objects named *guardians*. The reconfiguration of such objects happens when the system is in a consistent state. However, since Argus includes a operational system with support to transactions, a consistent state can always be obtained performing a rollback operation. Although this approach does not require adding explicit points of reconfiguration in the applications, providing support to rollbacks introduce a considerable overhead in the system.

There is also a proposal to add dynamic reconfiguration in the Conic system [KM90]. However, this proposal supports only structural reconfiguration. Recently, in [dP99] is presented another proposal to add dynamic reconfiguration in an agent based system. Since the agents in this proposal are not mobile, the focus is in structural and geometrical reconfiguration. Dynamic reconfiguration requiring module replacement is not handled by this proposal.

Stadel [Sta91] has proposed the introduction of dynamic reconfiguration in Eiffel applications. This proposal introduces in the Eiffel environment a dynamic linker and loader and also a configuration management utility, where the programmer can issue commands to dynamically replaces objects from a running application. Unlike the previous solutions described in this section, this proposal requires the participation of the user in the reconfiguration process.

7 Conclusions

Dynamic reconfiguration is a relevant requirement in many distributed systems. However, the works done until this moment in this area have focused mainly on applications designed following the traditional client/server model. In this paper, we have proposed an algorithm for dynamic reconfiguration of distributed applications designed in the mobile agent paradigm. We have also showed that the presented algorithm can be easily implemented in the IPL language. Particularly, handling objects using names with scope in the whole network and the blocking semantics used in method calls have contributed to make the implementation of the proposed algorithm straightforward.

Besides having mobile agent applications as its target, the proposed algorithm makes almost transparent the reconfiguration process. Unlike other proposals, programmers do not need to customize a script to guide the reconfiguration. However, the algorithm presented in this paper requires programmers to explicitly indicate the states of the application where a reconfiguration can take place. As further work, we intend to implement a version of IPL supporting the algorithm proposed in this paper.

References

[BD93] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: Theory and practice. *IEEE Software Engineering Journal*, 8(2):102–108, March 1993.

[Car95] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[Car99] Luca Cardelli. Abstractions for mobile computation. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer-Verlag, 1999.

[CG98] Luca Cardelli and Andrew Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[dP99] Noel de Palma. Dynamic reconfiguration of agent-based applications. Technical Report Project SIRAC, INRIA, 1999.

[Gor00] Andrew Gordon. Notes on nominal calculi for security and mobility. In *International Summer School on Foundations of Security Analysis and Design*, Bertinoro, Italy, September 2000.

[Hof93] Christine Hofmeister. *Dynamic Reconfiguration of Distributed Applications.* PhD thesis, Computer Science Department, University of Maryland, 1993.

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[LO99] Danny Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):88–89, 1999.

[Ope97] Open Group. DCE 1.1: Remote Procedure Call. Technical Report C706, Open Group, August 1997.

[Pur94] James Purtilo. The Polylith software bus. *ACM Transactions of Programming Languages and Systems*, 16(1):151–174, January 1994.

[Sta91] M. Stadel. Object oriented programming techniques to replace software components on the fly in a running program. *ACM SIGPLAN Notices*, 26(1):99–108, January 1991.

[VBLB00] Marco Túlio Valente, Roberto Bigonha, Antônio Alfredo Loureiro, and Mariza Bigonha. Object oriented languages with abstractions for mobile computation. In *Eletronic Notes on Theoretical Computer Science*, volume 38. Elsevier Science, 2000. (to appear).

[Whi97] James E. White. Mobile agents. In Jeffrey Bradshaw, editor, *Software Agents*, pages 437–472. AAAI Press/MIT Press, 1997.