A new mobile state protocol for Distributed Oz

M. Hadim & P. Van Roy
INGI-Université Catholique de Louvain
2 Place Sainte Barbe 1348 Louvain-la-Neuve Belgium
{pvr,muh}@info.ucl.ac.be

Abstract

One of the most important issue to address when designing an object-oriented language for distributed programming is related to object mobility. We present a new mobile state protocol for ensuring mutually exclusive access to mobile objects and cells in Distributed Oz. The protocol greatly improves the locality and scalability properties. It also allows potentially to overcome the problem of fault tolerance.

1 Introduction and Motivations

There are two conflicting goals in designing a language for distributed programming. First, the language should be network transparent, i.e., computations behave correctly independently of how they are partitioned among sites. Second, the language should give simple and predictable control over network communication patterns. The Distributed Oz language [14] satisfies these two goals. Its basic design principle is to distinguish clearly between the *language semantics* and the *distributed semantics*. Indeed, on one hand, it defines the language in terms of a language semantics, then a distributed semantics that refines it to take network behavior into account. On the other hand, it incorporates mobility in a fundamental way in the distributed semantics. This later extends the language semantics with *mobility control*. Namely, the ability for *stateful entities* to migrate between sites or to remain stationary at one site, according to the programmer's intention [14]. By stateful entities, we mean entities that change over time, i.e., they are defined by a sequence of states. Making mobility a primitive concept; i.e., at the language level, makes it possible to define efficient networked objects, whose mobility can be *precisely* controlled by the programmer. As it is well established [3, 9] that such concept is essential in distributed programming. Indeed many distributed systems support some concept of mobility, and the problem of designing efficient schemes for supporting access to shared object over a distributed system has been challenging. Stateful entities are mainly *cells*. Objects are defined in a straightforward way in terms of cells. Thus, in the reminder, we focus mainly on cells. Cells semantics is given in the next section. The system contains a *mobile state protocol* [14] that implements the distributed semantics of cells, while allowing the cell state to efficiently migrate between sites.

To emulate a shared memory on a distributed system, the mobile state protocol uses a *home-based* scheme [14]. Each cell is associated with a fixed site, termed as its *home site* and manages the cell access. Home-based schemes are simple and easy to implement. They have been observed to work well for small-to-medium scale systems [3]. Nevertheless, they suffer from problems of *scalability* and *locality*. As the number of sites grows, or if a cell is a "hot spot", that cell home site is likely to become a synchronization bottleneck, since it must mediate all access to the cell. Moreover, if a requesting client is far from the cell home, then it must incur the cost of communicating with the home, even if the site currently holding the state in nearby. Another problem arising with such schemes, is the crash of the home site. This situation induce the loss of the possibility to migrate the state between clients, even if the site holding it is alive.

One way to alleviate these problems is to introduce some concept of *redundancy and locality* on the set of sites which may potentially have access to the shared cell. In this paper, we propose, an original scheme which extends the mobile state protocol and allows to efficiently share cells between different nodes [4]. The main advantages of our protocol is that, first the communications necessary to migrate a cell state between sites, are minimized to a lower bound. Second, a nice locality/scalability property is integrated, and third, the proposed scheme allows to potentially overcome the problem of fault tolerance, by means of a redundancy concept, also introduced. The basic idea consist in organizing the network into *nested domains*.

In section 2, we give basic concepts and the mobile state protocol. In section 3, we present the new proto-

col, and give an useful improvement in section 4. We analyze the protocol in section 5. In section 6, we give related works, and conclude in section 7.

2 Basics

Distributed Oz is defined by transforming all its statements into statements of a small kernel language, called OPM (Oz Programming Model) [13]. OPM is a concurrent programming model with an interleaving semantics. Then, the semantics of Distributed Oz is obtained by given each basic entity of OPM a well defined distributed behavior. There are three network behaviors of OPM entities. We summarize below these distributed semantics:

1. Stateless entities: we mean entities that do not change over time. These entities are *replicated*, and the incidence of both the locality/scalability and fault tolerance properties on them is minor.
2. Single assignment entities: these are logic variables. The binding process is done by a *distributed unification protocol* [5], that implements their distributed semantics. The incidence of the locality/scalability and fault tolerance properties on them is of medium importance.
3. Statefull entities: these are cells, objects and threads. Threads reduction statements are done at the thread home site.

A cell is a mutable pointer that consists of two values : its name and content-edge. A cell is mobile and may be accessible from many sites. Its distributed semantics is detailed in the next section. An object is mobile, and its distribution semantics obeys its OPM definition. When a method is called remotely, it is replicated to the calling site, and executed over there. Subsequent method calls are local. When the object state is updated, the content-edge of the cell holding the state will migrate to the site; i.e., this is the distributed semantics of cells.

Cells are the unique entities which may held a state. Improving the locality/scalability property is most crucial for these entities. The operations on cells are creation, exchange, and access. We give the two semantics of the exchange. The semantics of the other operations can be found in [14], and are relatively more simpler.

Language semantics of Exchange. The exchange operation combines a read and a write operations. The part of the store that is not relevant to the rule is denoted by σ. This rule is reducible when its first argument refers to a cell name. It reduces to the new statement X=Z, which gives access to the old content Z through X. The content-edge is updated to refer to the new content Y.

Cell exchange

{Exchange C X Y}	X=Z
$\sigma \wedge C = n \wedge n : Z$	$\sigma \wedge C = n \wedge n : Y$

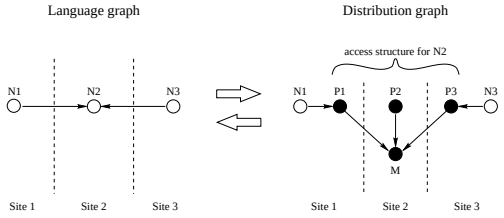

Figure 1: Language graph and distribution graph.

Distributed semantics of Exchange. We introduce the notion of a *representative* on a site. The representative of X on site i is denoted by X_i. Assuming that a cell with name n exists on sites 1,...,k, and that the content-edge is on site p with content Z, then the distributed reduction rule of the Exchange is:

$\{\text{Exchange C X Y}\}_q$	$(\text{X=Z})_q$
$C_1 = n... \wedge C_k = n \wedge (n : Z)_p \wedge$ $\forall i = 1...k, i \neq p : (n : \bot)_i \wedge \sigma_r$	$C_1 = n... \wedge C_k = n \wedge (n : Y)_q \wedge$ $\forall$i=1...k, i $\neq q : (n : \bot)_i \wedge \sigma_r$

The language graph. OPM programs execution are modeled using a graph transformation framework. This graph is the language graph and all its transformations respect the language semantics. Each entity corresponds to one node in this graph, and arrows correspond to references from (towards) an entity.

The distribution graph. The language graph is extended with the notion of *site*. We introduce a finite set of sites, and annotate each node of the language graph with a site. If a node N is referenced by a node on another site, then map it to a *set* of nodes. One node per site and one specific node responsible for all the others (see Figure 1). The resulting graph is called the **access structure** of N, and the operation that maps it into its access structure is called the *globalisation* of N. The graph resulting after all nodes have been globalised is the *distribution graph*. OPM distributed execution is also a sequence of this graph transformations.

The access structure notion. An access structure is the framework through which the distributed semantics of each entity is implemented. It has a single manager root node and a set of proxy nodes. The proxies are all on different sites. Each proxy points to its manager. The manager node has a unique global address. All access structures are constructed according to the same principle. Given a node N of the language graph, if a reference of this node leaves its site to a destination site d, then two scenarios are possible:
1. N's access structure exists yet: The reference of the manager node is transmitted to the destination site d. Thus, when the operation $import(N)$ is applied in d, the created proxy node gets a reference to its manager.
2. N is a local node : The globalisation operation creates an access structure for N. A manager node is

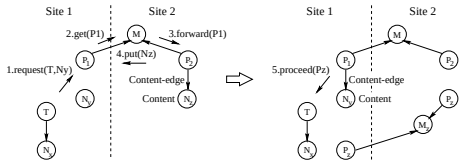

Figure 2: Distributed reduction of the Exchange.

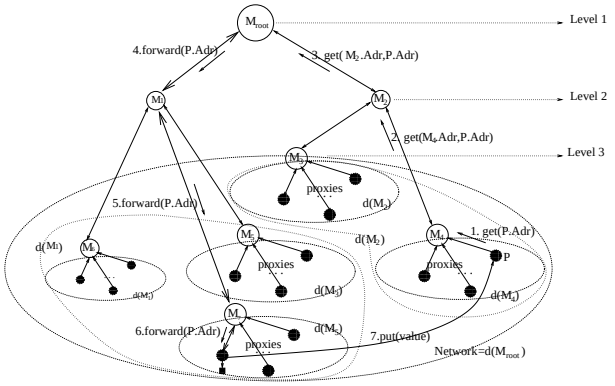

Figure 3: The new access structure

created in N's site, and its reference is transmitted to site d so as to retrieve similar conditions to case 1.

The mobile state protocol. This protocol [14] manages the movements of the cell's state. Each site has a proxy node. The manager node is located in the creation site of the cell and it has a pointer over the proxy holding the state (or will eventually hold it). When a proxy P needs the state to do an exchange, it sends the message $get(P.Adr)$ to the manager ($P.Adr$ being the address of P). The manager receiving this message, sends the message $forward(P.Adr)$ to the proxy P' currently holding the state. The manager therefore serializes multiples requests, guaranteeing that there is no starvation. Proxy P' receiving the forward message sends then the message $put(value)$ to the proxy P, if it already has the state. Otherwise, it stores the address of P, waits to receive the state and then sends it (see figure 2).

3 Extending the access structure

The new protocol consists in partitioning the network into *a tree of leveled domains*. Each domain is associated with a manager node, that handles the protocol implementing the distributed semantics of an entity, regarding the part d of the network. Indeed, a *managers tree* is constructed such that:

1. the root manager M_{root}, of level 1, handles the protocol over the network,
2. each manager M of level l and domain $d(M)$ admits n_c children $M_i, i = 1...n_c$, of level $l+1$ and subdomain $d(M_i)$ of $d(M)$. The union of all the domains $d(M_i), i = 1...n_c$ covers the domain d (see figure 3).

3.1 The new mobile state protocol

The managers tree is constructed. Now we specialize the discussion to cells. As a first step, assume that for a cell C a tree of managers has been created, where the leaf managers are of level tl. Suppose that a proxy P sited at site s executes an exchange (we assume the state not in s). P belongs to $d(M_i)$ domains, $i = 1...tl$ (M_1 being the root), and it has a reference to its father manager M_{tl}. To migrate the state to s, the following *recursive* scheme is observed:

P sends the message $get(P.Adr)$ to its father M_{tl} ($P.Adr$ being P's address). M_{tl} receives the message and checks wether the state is in its domain:

(a). If a proxy in $d(M_{tl})$ holds the state: M_{tl} sends the message $forward(P.Adr)$ to that proxy. Receiving this message, this later send then the message $put(value)$ to P. M_{tl} updates its state pointer to be $P.Adr$.

(b). If the state is not in $d(M_{tl})$: M_{tl} sends the message $get(M_{tl}.Adr, P.Adr)$ to its father M_{tl-1}, while updating its state pointer to be $P.Adr$. M_{tl-1} receiving the message, observes exactly the same treatment previously observed by M_{tl-1}. This illustrate the recursiveness of the scheme. As the state pointer of M_{root} is a non $NULL$ address, P's request is eventually satisfied.

Constructing the tree. We propose a *lazy* method for the tree construction. For a domain $d(M)$, M is created, if and only if there are *active proxies* in $d(M)$; i.e., proxies that have explicitly requested the state. Once M is created, the same process is refined for subdomains of $d(M)$, to create M's children. Let C be a cell created on site s_1. The first exportation over a site s_2 will create M_{root} on s_1 and its address is transmitted to s_2 (s_1's proxy stores M_{root} as being its father manager). As long as an exportation from s to s' is executed, the reference of M_{root} is transmitted to s'. Now, let $M_i, i = 1...m$ be the yet created children of a manager M. We assume that the union of $d(M_i), i = 1...m$ does not necessarily cover $d(M)$. When M receives a $get(P.Adr)$ message from P, it first looks for, in its *Sons* table, a son of the proxy's location subdomain, then:

1. If there is no son: M satisfies P's request regarding the state. It initiate the construction of a son (if $l < tl$): it sends the message $create\_manager(l+1)$ to P. It enters a *suspended* state, waiting for the address of its son to be created. P receives the message, creates a manager, say M', of level $l+1$ in its site and sets its father to be M', and send the message $update\_sons(M'.Addr)$ to M. M receives this message, suspends its suspended state, adds $M'.Adr$ to its $Son\_managers$ table, and updates its state pointer.
2. If there exists a son M': M sends the message

$manager(M'.Adr)$ to P. The latter receives the message, updates its father to be M' (P is thus attached to a nearest manager). M also sends the message $get(P.Adr, P.Adr)$ to M'. M' receives the message, and just satisfies P's request regarding the state.

3.2 Formal specification of the protocol

The protocol is defined as a set of nondeterministic reduction rules. We consider a set of manager and proxy nodes linked by a network N, that is a multiset containing messages of the form $d : m$ where d identifies a destination node, and m is a message. The initial configuration consists in the unique manager M_{root} with a set of proxies having a reference to it. We assume the state is at proxy P_1. Each node has state represented by a set of attributes. We denote attribute *Attribute* of node N by $N.Attribute$. Reduction rules are of the form: $\frac{\text{Condition}}{\text{Action}}$.

Execution follows an interleaving model. At each step, a rule with valid conditions is selected, then reduced atomically. A rule condition consists of boolean conditions on the node state and one optional receive condition, **Receive**(d,m), meaning that $d : m$ has arrived at d. Executing a rule with such a condition removes $d : m$ from the network and performs the action part of the rule. The action **Send**(d,m) asynchronously sends message m to node d, i.e., it adds the message $d : m$ to N. The network and the nodes are *fair*: Messages to any node take arbitrary finite time and arrive in arbitrary order. Any rule with valid conditions is eventually reduced. The table below lists the attributes of proxy and manager nodes.

Manager node
Adr: manager's address
Cc: pointer over the node containing the state
Fm: father manager's address
l: manager's level
Sons: the manager's existing set of son managers
Proxy node
State: initialized to CHAIN only for P_1
Thread: requesting thread
Addr: the address of the proxy
Forward: initialized to NULL for all proxies
Root_manager: initialized to M_{root} for all proxies
Fm: proxy's manager, initialized to M_{root} for all proxies
Content: initialized to N for P_1 and NULL otherwise
Newcontent: initialized to NULL for all proxies

Given a manager M receiving a $get(Addr)$ request, the decision function telling M wether a son manager is to be created, is obtained using, M's level and *Sons* table and the address *Addr*. This function is defined as follows:

$$locality(M, Addr) =$$

$$\begin{cases} \text{leaf if M.l=tl} \\ \text{negative if M.l} < \text{tl} \wedge \forall\ M'.Addr \in \text{M.Sons Addr} \notin \text{d(M')} \\ \text{M'.Addr if } \exists\ \text{M' | M'.Addr} \in \text{M.Sons} \wedge \text{Addr} \in \text{d(M')} \end{cases}$$

The managers tree is constructed. As for the informal description, we first assume the managers tree constructed and that each proxy is attached to a manager of level *tl*. We call $PR1$ the corresponding protocol. The rules below ensure the state migration:

Managers rules.

R1: $\left\{ \dfrac{\text{Receive(M,get(Addr))} \wedge \text{M.Cc} \neq \text{NULL} \wedge \text{locality(M,Addr)=leaf}}{\text{Send(M.Cc,forward(Addr))} ; \text{M.Cc} := \text{Addr}} \right.$

R2: $\left\{ \dfrac{\text{Receive(M,get(Addr))} \wedge \text{M.Cc=NULL} \wedge \text{locality(M,Addr)=leaf}}{\text{Send(M.Fm,get(M.Adr,Addr))} ; \text{M.Cc} := \text{Addr}} \right.$

R3: $\left\{ \dfrac{\text{Receive(M,get(Addrm,Addr))} \wedge \text{M.Cc=NULL}}{\text{Send(M.Fm,get(M.Adr,Addr))} ; \text{M.Cc} := \text{Addrm}} \right.$

R4: $\left\{ \dfrac{\text{Receive(M,get(Addrm,Addr))} \wedge \text{M.Cc} \neq \text{NULL}}{\text{Send(M.Cc,forward(Addr))} ; \text{M.Cc} := \text{Addrm}} \right.$

R5: $\left\{ \dfrac{\text{Receive(M,forward(Addr))} \wedge \text{M.Cc} \neq \text{NULL}}{\text{Send(M.Cc,forward(Addr))}\ \text{M.Cc} := \text{NULL}} \right.$

Proxy rules.

R1: $\left\{ \dfrac{\text{Receive(P,request(T,}N_y\text{))} \wedge \text{P.State=NULL}}{\text{Send(P.Fm,get(P.Adr))} ; \text{P.State:=CHAIN} ; \text{P.Thread:=T} ; \text{P.Newcontent:=}N_y} \right.$

R2: $\left\{ \dfrac{\text{Receive(P,request(T,}N_y\text{))} \wedge \text{P.Content} \neq \text{NULL}}{\text{Send(T,proceed(P.Content))} ; \text{P.Content:=}N_y\ ;} \right.$

R3: $\left\{ \dfrac{\text{Receive(P,put(}N_z\text{))}}{\text{P.Content:=}N_z\ ; \text{Send(P.Thread,proceed(P.Content))}\ \text{P.Thread:=NULL} ; \text{P.Content:=P.Newcontent}} \right.$

R4: $\left\{ \dfrac{\text{Receive(P,forward(Addr))}}{\text{P.Forward:=Addr}} \right.$

R5: $\left\{ \dfrac{\text{P.Forward} \neq \text{NULL} \wedge \text{P.Content} \neq \text{NULL}}{\text{Send(P.Forward,put(P.Content))}; P.State := NULL\ \text{P.Content:=NULL} ; \text{P.Forward:=NULL}i} \right.$

Outline of the proof. We give the proofs of the safety and liveness properties of PR1, using an invariant. For the simplicity sake, a proxy P is considered as a manager of level $tl + 1$ (we assume $(P.Adr \in (P.Fm).Sons)$). Let n_n be the number of all the tree nodes. The property T_1 representing the tree structure is defined as:

$$\begin{cases} T_1(M_{root}) \ \wedge \\ T_1(M_i) \equiv \exists!\ M_j \mid \begin{cases} M_i.Addr \in M_j.Sons \\ \wedge \quad M_i.Fm = M_j.Addr \end{cases} \end{cases}$$

We define the operation father as: Mj=father(Mi). The Tree set is: Tree = $\{M,\ T_1(M)\}$. We introduce the notion of a *path*. Intuitively, a path is a sequence of manager nodes such that, the first one is a proxy, and each node M_{i+1} points over M_i (and implicitly

the last node has, in N, a get message sent).

$\mathcal{P}_{1...k} = [M_1, ..., M_k], \mid \left\{ \begin{array}{l} M_1.State = CHAIN \wedge \\ M_{i+1}.Cc = M_i.Addr, i = 1.k \perp 1 \end{array} \right.$

The root path is: $\mathcal{P}_{root} = [M_1, ., M_{root}]$. On paths, we define the followings: • $first(\mathcal{P}_{1.k}) = M_1$ • $last(\mathcal{P}_{1.k}) = M_k$ • $set(\mathcal{P}_{1.k}) = \{M_1, ., M_k\}$. For a sequence of paths $[\mathcal{P}_1, ., \mathcal{P}_n]$, we define properties GW, GW' as:

$$GW([\mathcal{P}_1, ., \mathcal{P}_n]) \equiv$$

$$\left\{ \begin{array}{l} \forall i \in [1, ., n[\left\{ \begin{array}{l} last(\mathcal{P}_i).Adr : forward(first(\mathcal{P}_{i+1}).Adr) \in N \\ \oplus \ last(\mathcal{P}_i).Forward = first(\mathcal{P}_{i+1}).Adr \end{array} \right. \\ \wedge \\ last(\mathcal{P}_n).Fm : \left\{ \begin{array}{l} get(first(\mathcal{P}_1).Adr) \in N \quad \oplus \\ get(last(\mathcal{P}_n).Adr, first(\mathcal{P}_1).Adr) \in N \end{array} \right. \end{array} \right.$$

$$GW'([\mathcal{P}_1, ., \mathcal{P}_n]) \equiv$$

$$\left\{ \begin{array}{l} \forall \ i \in \ [1, ., n[\left\{ \begin{array}{l} last(\mathcal{P}_i).Adr \ : \ forward(first(\mathcal{P}_{i+1}).Adr) \in N \\ \oplus \ last(\mathcal{P}_i).Forward = first(\mathcal{P}_{i+1}).Adr \end{array} \right. \\ \wedge \quad \mathcal{P}_n = \mathcal{P}_{root} \end{array} \right.$$

Intuitively, $GW([\mathcal{P}_1, ., \mathcal{P}_n])$ means that manager $last(\mathcal{P}_n)$ has sent a get request to its father, asking for the state requested by the proxy $first(\mathcal{P}_1)$ (this is the path $\mathcal{P}_n$), and there is a forward message toward each manager $last(\mathcal{P}_i)$, asking for the state requested by the proxy $first(\mathcal{P}_{i+1})$. GW' differs from GW only by the fact that the path $\mathcal{P}_n$ is the root path $\mathcal{P}_{root}$.

By extension the operation $set([\mathcal{P}_1, ., \mathcal{P}_n])$ is defined as: $\bigcup_{i=1}^{1=n} set(\mathcal{P}_i)$. Then, let SP be the set of all paths in the managers tree in some configuration C_i. The invariant I_1 of protocol $PR1$ is defined as:

$I_1 = I_{t_1} \wedge I_{sp_1} \wedge I_{gw_1} \wedge I_{rest_1} \wedge I_{put} \wedge I_u$, where,

$I_{t_1} \equiv \ T_1(M_i), i = 1...n_n \ ; I_{sp_1} \equiv SP = SP_1 \uplus \biguplus_{i=2}^{n} SP_i$

$I_{gw_1} \equiv \wedge_{i=2}^{n} GW(SP_i) \ \wedge \ GW'(SP_1)$

$$I_{put} \equiv \left\{ \begin{array}{l} SP_1 = [\mathcal{P}_1, ., \mathcal{P}_{root}] \wedge \\ \left\{ \begin{array}{l} first(\mathcal{P}_1).Content = NULL \wedge \\ first(\mathcal{P}_1).Adr : put(VALUE) \in N \\ \oplus \ first(\mathcal{P}_1).Content = VALUE \end{array} \right. \end{array} \right.$$

$I_{rest_1} \equiv \forall \ M \in (Tree \perp \bigcup_{i=1}^{n} set(SP_i))$

$\left\{ \ M.Cc = NULL \ \oplus \ M.State = NULL \ \right\}$

$I_u \equiv \{$ *all messages in N are unique and explicitly mentionned in* $I_1 \}$

Intuitively, I_{t_1} means that the nodes form a tree. I_{sp_1} means that SP is partitioned in configuration C_i into, one SP_1 set of paths satisfying property GW', and all the other set of paths satisfy property GW. I_{put} means that the put message will traverse the chain $[first(\mathcal{P}_1), ..., first(\mathcal{P}_{root})]$. I_{rest_1} means that all the nodes that are not in a path, are in a NULL state. Then, the following theorems give the safety and liveness properties of protocol $PR1$.

Theorem 1 *Formula I_1 is an invariant of $PR1$.*

Theorem 2 *Assuming fairness assumptions, requesting the state by a proxy node will cause it eventually to arrive once.*

Constructing the managers tree. To construct the managers tree, we add the following rules:

Managers rules added.

R6:
Receive(M,get(Addr)) ∧
locality(M,Addr)=Addr1
———————————
Send(*Addr*, *manager*(*Addr*1))
Send(*Addr*1, *get*(*Addr*, *Addr*))

R7:
Receive(M,get(Addr)) ∧ M.Cc=NULL
∧ locality(M,Addr)=negative
———————————
Send(M.Fm,get(M.Addr,Addr))
Send(Addr,create_manager(M.l+1))
M.Cc := suspended

R8:
Receive(M,get(Addr)) ∧ M.Cc≠ NULL ∧
locality(M,Addr)=negative
———————————
Send(M.Cc,forward(Addr))
Send(Addr,create_manager(M.l+1))
M.Cc := suspended

R9:
Receive(M,update_sons(Addr))
———————————
Add_son_managers(Addr)
M.Cc := Addr

proxy rules added.

R6:
Receive(P,manager(Addr))
———————————
P.Fm:=Addr

R7:
Receive(P,create_manager(l))
———————————
Addr:=Create_manager(l,P.Fm)
Send(P.Fm,update_sons(Addr))
P.Fm:=Addr

Outline of the proof. We first consider adding rule 6 of the proxies and managers, and show that these rules allow to attach each proxy with the manager of level tl of its domain. We call $PR2$ the corresponding new protocol. We redefine property GW as follows:

$$\mathrm{GW2}([\mathrm{P}_1, ., \mathcal{P}_n]) \equiv$$

$$\left\{ \begin{array}{l} \forall i \in \ [1, ., n[\\ \left\{ \begin{array}{l} last(\mathcal{P}_i).Addr : \ forward(first(\mathcal{P}_{i+1}).Addr) \in N \\ \oplus \ last(\mathcal{P}_i).Forward = first(\mathcal{P}_{i+1}).Addr \end{array} \right. \\ \wedge \\ \left\{ \begin{array}{l} \left\{ \begin{array}{l} (father(last(\mathcal{P}_n))).Addr : \\ \left\{ \begin{array}{l} get(first(\mathcal{P}_1).Addr) \in N \\ \oplus \\ get(last(\mathcal{P}_n).Addr, first(\mathcal{P}_1).Addr) \in N \end{array} \right. \\ \wedge \\ \left\{ \begin{array}{l} first(\mathcal{P}_n).Addr : manager(M.Addr) \notin N \\ \vee \ (first(\mathcal{P}_n).Fm \neq M.Addr) \end{array} \right. \end{array} \right. \\ \oplus \\ \left\{ \begin{array}{l} \exists \ M \mid M.Addr \in (last(\mathcal{P}_n).Sons \wedge \\ M.Addr : get(last(\mathcal{P}_n).Addr, first(\mathcal{P}_1).Addr) \in N \\ \wedge \\ first(\mathcal{P}_n).Addr : manager(M.Addr) \in N \end{array} \right. \end{array} \right. \end{array} \right.$$

The new invariant is: $I_2 = I_{t_1} \wedge I_{sp_1} \wedge I_{gw_2} \wedge I_{rest_1} \wedge I_{put} \wedge I_u$, where, $I_{gw_2} \equiv \wedge_{i=2}^{n} GW2(SP_i) \wedge GW'(SP_1)$. Then we have the followings:

Theorem 3 *Formula I_2 is an invariant of $PR2$.*

Theorem 4 *Let M_4 be a proxy attached to a manager M_1 of level $l < tl$ in configuration C_i. Assuming, the*

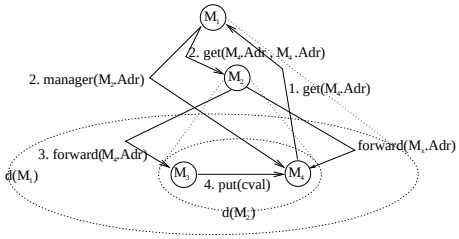

Figure 4: Assumption regarding managers rule 6

conditions of figure 4 and fairness assumptions, then reducing rule 1 *by* M_4 *allows:* (1) *to attach* M_4 *to a manager* $M_2 \in Sons(M_1)$*, of level* $l+1$*, and domain* $d(M_2)$ *containing address* $M_4.Adr$*, and*
(2.) *the state to be forwarded to proxy* M_4*.*

Finally, we add all the rules. Let $PR3$ be the protocol thus obtained. We show that the managers tree is constructed and that each time a proxy requests the state, it will eventually receive it. We first clarify the procedure $Create\_manager(level, Fathadr)$. Executing this procedure by a proxy M_1, creates a manager M_2 in the proxy's site. The level *level* and the proxy's site allow to infer an address ADR for M_2 and the following updates are done atomically: $M_2.Adr = ADR, M_2.Cc = M_1.Adr, M_2.Fm = M_1.Fm, M_2.l = level, M_2.Sons = NULL, M_1.Fm = ADR$. For the last invariant, we redefine property T_2 as follows:

$$\left\{\begin{array}{l} T_2(M_{root}) \wedge \\ T_2(M_i) \equiv \exists!\ M_j \mid \\ \left\{\begin{array}{ll} \left\{\begin{array}{c} M_i.Adr \in (M_j).Sons \oplus \\ M_j.Adr : update\_sons(M_i.Adr) \in N \end{array}\right. & \\ \wedge \quad M_i.Fm = M_j.Adr & \end{array}\right\} \end{array}\right.$$

The new Tree set is: $Tree = \{M \mid T_2(M)\}$. As there are node creation messages, we redefine a path as:

$$\mathrm{P2}_{1..k} = P_{12} + [M_3, ., M_k],\ \mid$$
$$\left\{\begin{array}{l} M_1.State = CHAIN \\ \wedge \left\{\begin{array}{l} \left\{\begin{array}{l} M_2.Cc = M_1.Addr \wedge P_{12} = [M_1, M_2] \\ \oplus \\ M_2.Cc = suspended\ \wedge \\ \left\{\begin{array}{l} \left\{\begin{array}{l} M_1 : create\_manager(M_2.l+1, M_2.Addr) \in N \\ \wedge\ P_{12} = [M_1, M_2] \end{array}\right. \\ \oplus \\ \left\{\begin{array}{l} M_2 : update\_sons(Addr) \in N \wedge \\ P_{12} = [M_1, M_1', M_2]\ \wedge\ M_1'.Cc = M_1.Addr \\ \wedge\ M_1'.Addr = Addr \end{array}\right. \end{array}\right. \end{array}\right. \\ M_{i+1}.Cc = M_i.Addr, i = 2...k \perp 1 \end{array}\right.$$

The last invariant is defined as:
$I_3 = I_{t_2} \wedge I_{sp_2} \wedge I_{gw_3} \wedge I_{rest_1} \wedge I_{put} \wedge I_u$, where $I_{t_2} \equiv T_2(M_i), i = 1...n_n$, and I_{sp_2}, (resp. I_{gw_3}) are defined as I_{sp_1} (resp. I_{gw_2}). The following theorems give the safety and liveness properties of protocol $PR3$.

Theorem 5 *Formula* I_3 *is an invariant of* $PR3$*.*

Theorem 6 *Let a proxy* M_1 *attached to a manager* M_2 *such that* $M_1.Addr \notin d(M_3), \forall M_3.Addr \in M_2.Sons$*, in configuration* C_i*. If* $M_2.l < tl$ *and fairness assumptions are ensured, then reducing rule* 1 *by* M_1 *allows:*
1. To create a son manager M_3 *of* M_2 *whose domain* $d(M_3) \subset d(M_2)$ *contains the address* $M_1.Adr$ *with the right updates, and,*
2. The cell content to be forwarded to M_1*.*

4 An useful improvement

In this section, we improve the new mobile state protocol regarding the suspension state of the manager nodes, when there are creation of their children. Indeed, we would allow a manager M to treat the requests it receives between the duration issued, since it assigns the value "*suspended*" to its attribute Cc and until it receives the $updtate\_sons(Address)$ message. When M is in such a state, we consider the possible treatment of solely the messages $get(Adrman, Adress)$ or $forward(Adress)$ sent respectively by one of its children managers or father manager. For this purpose, we add a new attribute *"Forward"*, initialized to $NULL$, to manager nodes and the following two rules:

$$\text{R10:}\left\{\begin{array}{l} \text{Receive(M,get(Addrman,Address))} \wedge \\ \text{M.Cc=suspended} \\ \hline \text{M.Forward := Address} \\ \text{M.Cc := Addrman} \end{array}\right.$$

$$\text{R11:}\left\{\begin{array}{l} \text{Receive(M,forward(Address))} \wedge \\ \text{M.Cc=suspended} \\ \hline \text{M.Forward := Address} \\ \text{M.Cc = NULL} \end{array}\right.$$

Namely, when M receives one of the above messages, the attribute *"Forward"* is then used to store the address $Address$ of the requesting proxy, and the protocol simply continues to operate as usual. When manager M receives the $update\_sons(Address)$ message, the request of the proxy whose address is contained in the *"Forward"* attribute is then satisfied. Indeed, rule 9 is changed in the following two rules that handle this situation:

$$\text{R9a:}\left\{\begin{array}{l} \text{Receive(M,update\_sons(Address))} \wedge \\ \text{M.Forward=NULL} \\ \hline \text{Add\_son\_managers(Address)} \\ \text{M.Cc = Address} \end{array}\right.$$

$$R9b:\left\{\begin{array}{l} \text{Receive(M,update\_sons(Address))} \wedge \\ \text{M.Forward} \neq \text{NULL} \\ \hline \text{Add\_son\_managers(Address)} \\ \text{Send(Address, forward(M.Forward))} \\ \text{M.Forward = NULL} \end{array}\right.$$

Then, to avoid coherence problems that may arise because of concurrent creations of children managers of a manager M, rules 7 and 8 are also respectively changed as follows:

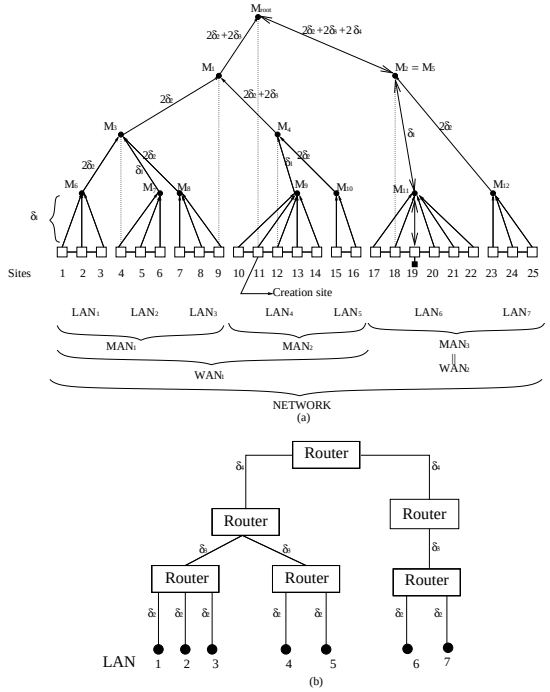

Figure 5: An illustration of the protocol

R7: { Receive(M,get(Address)) ∧ M.Forward=NULL ∧ (locality(M,Address)=negative) ∧ M.Cc=NULL

Send(M.Fm,get(M.addr,Address))
Send(Address,create_manager((M.l +1),M.addr))
M.Cc := suspended

R8: { Receive(M,get(Address)) ∧ M.Forward=NULL ∧ M.Cc=Addr ∧ locality(M,Address)=negative
---
Send(M.Cc,forward(Address))
Send(Address,create_manager((M.l +1),M.addr))
M.Cc = suspended

In summary, we just use the attribute *"Forward"* as a buffer variable that allows to avoid the strong synchronization between a manager and its children being created. The safety and liveness properties of the protocol, with the newly added rules, continue to hold.

5 Analysis of the protocol

We give some characteristics of the new protocol. We consider a network interconnected as in figure 5(b), and a hierarchical subdivision of it into: two WANs (Wide Area Network), tree MANs (Medium Area Network) and 7 LANs (Local Area Network). We assume δ_1 (resp. δ_2, δ_3 and δ_4) is uniformly the transmission delay within a LAN (resp. MAN, WAN and between two WANs). Let C be a cell created at site 11, whose access structure is as in figure 5(a). We compare the delay necessary to migrate the state, in the old and new protocols. We measure the ratio: $\rho = \frac{Migration\ delay\ in\ the\ new\ protocol}{Migration\ delay\ in\ the\ old\ protocol}$. The table below gives some values of ρ for some migration scenarios. "Within LAN_i" (resp. "MAN_i") means that the state is migrated between two sites of LAN_i (resp. MAN_i). As it brings out, migration delays are shorter in the new protocol.

	ρ
Within LAN_1	$\frac{(3\delta_1)}{(4\delta_2+4\delta_3+\delta_1)} << 1$
Within LAN_4	$\frac{(3\delta_1)}{(3\delta_1)} = 1$
Within LAN_5	$\frac{(3\delta_1)}{\delta_1+4\delta_2} << 1$
Within LAN_6	$\frac{(3\delta_1)}{(\delta_1+2\delta_2+2\delta_3+2\delta_4)} << 1$
Within MAN_1	$\frac{(\leq(6\delta_2+2\delta_1))}{(6\delta_2+4\delta_3)} << 1$
Within MAN_2	$\frac{(4\delta_2+3\delta_1)}{(\delta_1+4\delta_2)} \cong 1$
Within MAN_3	$\frac{(3\delta_1+4\delta_2)}{(6\delta_2+4\delta_4+4\delta_3)} << 1$

The new protocol improves the locality property in that sense that migrating the state within a domain is managed *locally* by the corresponding manager. As it improves the scalability property. In fact, the proxies are not attached to a single manager which may be overloaded when treating their several requests. Rather, the charge of these requests is spread over several managers; each responsible for a subset of the whole set. The managers tree act indeed as a hierarchy of cache memories that enables to make remote references only if the state is not "within the cache": within the domain of the manager. When constructing the tree, an improvement may be to create a manager for a domain d only if there is a persistent request from d; i.e., only if a constant c get requests are made from d. This is handled simply. It just suffices to add a control variable. For example, a new manager's rule 7 is:

R7: { Receive(M,get(Address)) ∧ M.Cc=NULL ∧ locality(M,Address)=negative)

Send(M.Fm,get(M.Addr,address))
counter=counter+1
IF counter(d) = c THEN
  $Send(Address, create\_manager((M.l+1), M.Addr))$
  $M.Cc = suspended$

To allow to attach a proxy directly to its greatest level manager, when a globalisation operation is performed from site s to s', the proxy of s may transmit the reference of its father manager M to the created proxy of s', if s' is in $d(M)$.

The new protocol allows potentially to overcome efficiently the problem of fault tolerance. Namely, the redundancy concept introduced can be used as a basic mechanism to tolerate sites failures.

6 Related works

Many systems, that we know of, except Emerald [7] and Obliq [2] do distributed execution by adding a distribution layer on top of a centralized language, i.e., CORBA [11], Erlang [15], Java [9]. This has the disadvantage that distribution is not a seamless extension to the language. In Emerald, objects are stationary

by default and explicit primitive operations exist to move them. Moving a mutable object is an atomic operation that clones the object on the destination site and aliases the original object to it. The result is that messages to the original object are passed to the new object through an aliasing indirection. This induce aliasing chains. Obliq has taken a first step toward the goal of conservatively extending language entities to a distributed setting. Obliq objects are stationary. Object migration in Obliq can be implemented in two phases by cloning the object on another site and by aliasing the original object to the clone. The migration procedure must be executed internally. The result is an aliasing chain too.

Small-scale systems typically uses a broadcast-based protocol to locate objects in a distributed system of caches (proxies). Existing large-scale systems are either home-based, or use a combination of home-based and aliasing pointers [1, 6, 10, 8]. A closely related protocol is the Arrow Distributed Directory Protocol described in [3]. According to the authors, this protocol allows a scalable and local mechanism for ensuring mutually exclusive access to mobile objects. This protocol is given by a minimum spanning tree T over the network, where each node having potentially access to the object, stores a link in T, arising on the shortest path to get the object. This protocol ensures a locality property; however, it is not at all clear, how it is scalable. When a node is added to the set of nodes having access to the object, how this node is integrated in T ?. If a new spanning tree is re-computed, this is very costly. Plaxton et al. give in [12] a randomized directory scheme for read-only object.

7 Conclusion

We have presented a new design for constructing access structures for Distributed Oz entities. This design aims to improve the locality and scalability properties.

As we have presented a new mobile state protocol for mutually exclusive access to cells and objects in Distributed Oz. As an extension of the old protocol and in the philosophy of Distributed Oz design, the new protocol is integrated at the semantical level; the proxy and manager rules implement the distributed semantics of cells. The new protocol greatly improves the locality/scalability properties, allowing efficient state migration delays and spreading uniformly the charge of the proxies requests.

In a more general context, the protocol described in this paper can be used in client-server architectures in order to improve the locality/scalability properties. One can imagine a tree of servers spread among a distributed network that serve client requests for some services. Each client sending its request to a "nearest" server. Some cooperative work between the servers may then be necessary to maintain some coherent state between them.

Acknowledgements

This research is partly financed by the Walloon Region of Belgium. Raphaël Collet gave many valuable comments on this article.

References

[1] B Bershad, M Zekauskas, and W.A Swadon. The midway distributed shared memory systeml. In *Proceedings of 38th IEEE Computer Society International Conference.*, pages 152-164 jun 1997.

[2] Luca Cardelli. A language with distributed scope. *ACM Transactions on Computer Systems*, 8(1):27–59, January 1995.

[3] M.J Demmer and M.P Herlihy. The arrow distributed directory protocol. In *Proceedings of 12th International Symposium on Distributed Computing. LNCS 1499*, Greece Sep 1998.

[4] M Hadim and P Van Roy. A locality/scalability model for Distributed Oz. *Research report RR 99-03*, INFO - Université Catholique de Louvain, April 1999.

[5] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *TOPLAS*, To appear.

[6] K.L Johnson, M.F Kaashoek, and D.A Wallach. Crl: High-performance all-software distributed shared memmory. In *Proceedings of 15 ACM Symposium on Operating Systems Principles.*, pages 213-228 Dec 1995.

[7] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[8] P Keleher, S Dwarkadas, A.L Cox, and W Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *In Proc. of the Winter 1994 USENIX Conference.*, pages 115-131 Jan 1994.

[9] Sun Microsystems. *The Java Series.* Sun Microsystems, Mountain View, Calif., 1996.

[10] R.S Nikhil. Cid: A parallel, "shared memory" c for distributed-memory machines. In *Proc of 7th International Workshop on Languages and Compilers for Parallel Computing.*, Aug 1994.

[11] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA: The Common Object Request Broker Architecture.* Prentice-Hall PTR, Upper Saddle River, N.J., 1996.

[12] C.G Plaxtron, R Rajaman, and A.W Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of 9th Annual ACM Symposium on Parallel Algorithms and Architecturs.*, pages 311-321 Jun 1997.

[13] Gert Smolka. The Oz programming model. In *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[14] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *TOPLAS Vol. 19, No. 5*, Sep 1997.

[15] Claes Wikström. Distributed programming in Erlang. In the *1st International Symposium on Parallel Symbolic Computation (PASCO 94)*, pages 412–421, Singapore, September 1994. World Scientific.