Path Redundancy in a Mobile-State Protocol as a Primitive for Language-Based Fault Tolerance

Per Brand[*] Peter Van Roy[†] Raphaël Collet[‡] Erik Klintskog[§]

January 17, 2000

Contact author Raphaël Collet, `raph@info.ucl.ac.be`
Format Regular paper

Abstract

Mobile-state protocols are important for distributed object systems. We define a lightweight mobile-state protocol that has a well-defined behavior for site failures and network inactivities. It uses a redundant copy of the path defined by the state movements in the network to manage failures along that path. The protocol is implemented as part of the Mozart platform for distributed application development based on the Oz 3 language. The protocol provides enough information to the language layer so that we can use the platform to program common fault-tolerant algorithms completely in Oz.

[*] `perbrand@sics.se`, Swedish Institute of Computer Science, S-164 28 Kista, Sweden
[†] `pvr@info.ucl.ac.be`, Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium
[‡] `raph@info.ucl.ac.be`, Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium
[§] `erik@sics.se`, Swedish Institute of Computer Science, S-164 28 Kista, Sweden

1 Introduction

1.1 The problem

Distributed computing has to deal with several distribution-specific issues, such as latency and *partial failures* [WWWK94]. Those concern computers (hardware and software) and network links. In spite of that, a distributed application should continue to work or at least, it should not produce incorrect results.

A programming language with a semantics that takes distribution and partial failures into account is of major interest for programmers. They only have to know how faults are reflected *at the language level*. Fault detection and report can be used as a foundation for programming fault-tolerance [Gär99]. Oz 3 is such a language [HF99, HVBS98], and is the base of the Mozart system [Moz99], a platform for general-purpose distributed application development. Mozart combines efficient network-transparent distribution with a failure model that allows programming fault-tolerant behavior in Oz [VHB99, AV00].

The subject of this paper is the implementation of distributed objects, particularly *mobile objects*, with a well-defined fault behavior. A mobile object behaves as a cache: its state migrates to a site invoking a method, so that the method is performed locally (at the caller's site).

CORBA [OMG99], DCOM and RMI [Sun98] provide distributed objects, but none of them provide mobility as a primitive for distribution. Mobile object protocols exist, such as the Arrow Distributed Directory Protocol [DH98], but they are difficult to adapt for system with partial failures.

1.2 Our contribution

Our aim in this paper is to design a mobile-state protocol that has a well-defined behavior for site failures and network inactivities. The protocol is part of the Mozart system, and it allows to implement fault-tolerant algorithms at a higher level of programming.

Actually, the protocol implements a *cell*, i.e., a distributed mutable pointer that is the basic stateful entity at the heart of the Oz object system. It must provide an atomic exchange operation on that pointer, and is subject to the following requirements.

Requirement 1. The protocol must provide a *thread interface*, by which any thread can request a pointer exchange and be notified of the success or the failure of the operation. The returned information must be reliable.

Requirement 2. The protocol must be built on top of a *failure model* that covers the vast majority of failures occuring in general-purpose distributed systems.

Requirement 3. The protocol must use the same *mobility strategy* as the non-fault-tolerant mobile-state protocol of [VHB$^+$97] in the case of no failure. The mutable pointer moves from site to site, and the value exchange is performed locally.

Requirement 4. The protocol does not sacrifice *performance* in the common case of no failure. More precisely, it has the same message latency as the protocol of [VHB$^+$97].

The latter is a central idea in the design of the protocol: the degree of fault tolerance provided is minimal if nothing special is done at the language level, but the protocol is *efficient*. For that reason, it does *not* provide *state recovery* by managing some state redundancy. Rather, it is able to *recover the path that the pointer follows through the network*, by managing a certain *path redundancy*.

1.3 Outline of the paper

The paper is organized as follows. Section 2 presents a formal model for a network of computers on which the protocol is built. The model includes the definition of failures. Sections 3, 4, 5 and 7 define the protocol itself. It is presented step-by-step, from a basic non-fault-tolerant protocol to the final one. Section 6 explains how we prove the protocol correct. Section 8 concludes the paper.

2 The site and network model

This section defines the model on which the protocol is built. It is intended to provide abstractions for distributed applications over the Internet and TCP-based networks. Section 2.2 introduces the notions of *site*, *network* and the possible failures to be modelled. Section 2.3 shows how network operations can be performed within the model. Then the adequation between the model and actual systems is discussed in section 2.4.

2.1 Execution model and notations

The protocol is described with *condition-action rules* that are executed *atomically.* Each rule stands as a guarded command in a loop. The semantics of a set of rules is given by the equivalence

$$\frac{condition_1}{action_1} \cdots \frac{condition_k}{action_k} \quad \equiv \quad \begin{array}{cl} \mathbf{do} & condition_1 \longrightarrow action_1 \\ [] & \cdots \\ [] & condition_k \longrightarrow action_k \\ \mathbf{od} & \end{array}$$

Condition-action rules contain *global* and*local* variables. Global variables are written in roman style and constitute the state of the system. Local variables are written in italic and their scope is limited to one rule. If they appear in the condition part of a rule, they are automatically instanciated when the rule is applied. The assignment operator is noted $\leftarrow$.

A rule whose condition is evaluated to **true** is said to be *executable.* By default we assume that the execution is *strongly fair*, which means that a rule that is infinitely often executable will eventually be executed.

2.2 Sites and network

Distribution model. The system consists of a set of nodes communicating through a network. Each node is like a process with a unique *node identifier* N_i, several attributes (its own state), and it executes condition-action rules. The network is a multiset of messages of the form $N : M$, where N identifies the destination node and M is the message content. The network is assumed to be asynchronous and unordered (i.e., non-FIFO).

We introduce distribution by partitioning the set of nodes among a set of *sites.* In order to achieve this, we use a set of *site identifiers* $S_1, \ldots, S_m$, and a *site function* such that $\mathrm{site}(N)$ gives the site of node N. We also partition the network among the sites. Each site S has its own *network neighbourhood* $\mathrm{net}(S)$, which can model network buffers.

In Figure 1 we can see two nodes N_1 and N_2 on different sites. The picture also shows how node N_1 can send a message to node N_2. The message follows the arrows labelled by "Send", "Transfer" and "Receive". Each arrow represents an atomic transition; they are described in Section 2.3.

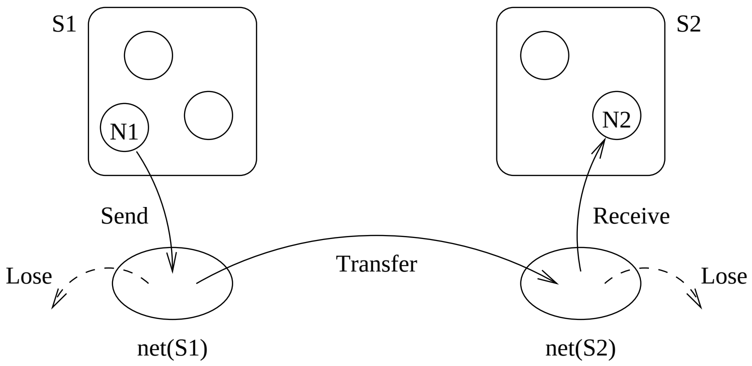


Figure 1: The site and network model, showing possible atomic operations.

Failure model. Now we formally represent failures. Our model admits two kinds of failure: site failures and network inactivities. It also includes a detection mechanism.

- A *site failure* is a permanent inactivity of a site. For each site S, a boolean variable $\mathrm{up}(S)$, initially **true**, becomes **false** when a site fails, and it remains **false** forever.
- A *network failure* is a possibly temporary inactivity of the network between two sites. For each site pair $\langle S, S' \rangle$, a boolean variable $\mathrm{active}(S, S')$, initially **true**, can transit back and forth between **true** and **false**. The network link between sites S and S' is functional when $\mathrm{active}(S, S')$ is **true**.
- A *detection* variable $\mathrm{detectsf}(S, S')$[1] indicates if site S already detected whether site S' has failed. That variable is initially **false** and becomes **true** when the site failure is detected. Note that the detection mechanism must be *reliable*: a site S detects that site S' failed *only if* that site S' actually failed.

The following rules are part of the model and define how failures happen and how they are detected. Variables S and S' can be any site identifiers.

S.1 Site failure

$$\frac{\mathrm{up}(S)}{\mathrm{up}(S) \leftarrow \textbf{false}}$$

S.2 Network activity

$$\frac{\textbf{true}}{\mathrm{active}(S, S') \leftarrow \neg\mathrm{active}(S, S')}$$

S.3 Site failure detection

$$\frac{\mathrm{up}(S) \wedge \mathrm{active}(S', S)}{\mathrm{detectsf}(S, S') \leftarrow \neg\mathrm{up}(S')}$$

The fairness assumptions about those rules are discussed in Section 2.4.

2.3 Network operations

We provide the model interface with two operations (**send**() and **receive**()) and three failure tests (**at**(), **sitedown**() and **netdown**()). We also define two internal operations for transmitting messages, in the no-failure case and in the failure case.

Internal operations. The system provides operations for the transmittion of messages from site to site. These operations are represented by the arrows labelled "Transfer" and "Lose" in Figure 1, and they are formally described by the following rules.

[1] "detectsf" stands for "detect site failure".

S.4 Transfer message

$$\frac{N' : M \in \text{net}(S) \land S' = \text{site}(N') \land \text{up}(S) \land \text{active}(S, S')}{\begin{array}{l}\text{net}(S) \leftarrow \text{net}(S) \ominus \{N' : M\} \\ \text{net}(S') \leftarrow \text{net}(S') \oplus \{N' : M\}\end{array}}$$

S.5 Lose messages

$$\frac{\text{net}(S) \neq \emptyset \land \neg\text{up}(S)}{\text{net}(S) \leftarrow \emptyset}$$

Note. We use operations $\oplus$ and $\ominus$ on multisets. They represent the disjoint union and the difference between multisets, respectively. The empty multiset is represented by $\emptyset$.

Network interface. We provide two operations and three predicates for user algorithms. They can be viewed as usage facilities. They allow model variables not to appear in user rules.

The effect of the operation **send**() (executed at node N) is to put the given message into the local network of the sender (1). The operation **receive**() (executed at node N) stands for *both* a condition and an action. A rule containing such an operation can be rewritten as shown in (2).

$$\mathbf{send}(N' : M) \equiv \begin{array}{l} S \leftarrow \text{site}(N) \\ \text{net}(S) \leftarrow \text{net}(S) \oplus \{N' : M\} \end{array} \tag{1}$$

$$\frac{\mathbf{receive}(N' : M) \land \mathit{condition}}{\mathit{action}} \equiv \frac{S = \text{site}(N) \land N' : M \in \text{net}(S) \land \mathit{condition}}{\begin{array}{l}\text{net}(S) \leftarrow \text{net}(S) \ominus \{N' : M\} \\ \mathit{action}\end{array}} \tag{2}$$

These atomic operations are represented by the arrows labelled "Send" and "Receive" in Figure 1.

The predicate **at**() is a site activity test that is **true** iff the site of the given node is up (3). It also indicates at which node the execution must take place. The predicate **sitedown**(), when executed at node N, evaluates to **true** if the given node has been detected as down (4). The predicate **netdown**() is a network activity test. When executed in node N, it indicates if another node is currently unreachable (5).

$$\mathbf{at}(N) \equiv \text{up}(\text{site}(N)) \tag{3}$$

$$\mathbf{sitedown}(N') \equiv \text{detectsf}(\text{site}(N), \text{site}(N')) \tag{4}$$

$$\mathbf{netdown}(N') \equiv \neg\text{active}(\text{site}(N), \text{site}(N')) \tag{5}$$

2.4 Discussion

In this section we argue why the model is close to actual systems, for instance systems with a TCP-based network. We particularly consider site and network failures.

Site failures. When a site fails, every activity on it is definitively stopped. So the site crashes and it never respawns. This is very close to a process kill in the operating system, and it suits perfectly to our implementation of the Mozart platform [Moz99].

The good property with a site failure is that other sites *can* know the failure is permanent. That property strongly contrasts with temporary failures, which are characterized by an absence of knowledge. Of course, the detection of a permanent failure can be prevented by other failures, e.g., if a site is unreachable for a long period of time. The detection variables and the rule S.3 reflect that phenomenon.

Network behavior. The network model has been designed to correspond to the *Reliable Message Layer* of the platform, which is currently built on TCP [HVS97]. Yet it is surprising that a so-called "reliable protocol" can still result in message loss. But this is due to the way that TCP communicates with the source and destination sites.

The network neighbourhoods $\mathrm{net}(S)$ model the buffers at the sending and receiving sites. What TCP guarantees is the atomic transition between the send and receive buffers, i.e., there is no effect from intermediate nodes (pure end-to-end). However, the receipt and the reduction of a rule are not simultaneous: the message is in the destination process before it reduces a rule. So a message could have been correctly transmitted, then lost because the destination site failed before it can process it. Our model mimics that behavior.

Actually, a network inactivity can model many unknown remote problems. Such problems are a temporary network inactivity because of a delay, for example; a permanent network failure, where the system cannot guarantee that the network will ever become active again; a site failure that has not been diagnosed yet.

Fairness assumption about failure rules. The rules S.1, S.2 and S.3 above define how failure advents are modelled. We examine the strong fairness assumption on each of those rules.

Assuming a strongly fair execution of rule S.1 means that every site will eventually fail. In this paper, we do not make that assumption; strong arguments can be made for doing it, though. The strong fairness assumption on rule S.2 implies that network inactivities are never permanent. Making that assumption depends on whether the model should include permanent network failures. If rule S.3 is strongly fair executed, then any site failure will eventually be detected by other sites, provided network failures are never permanent. In this paper, we make that assumption, which means that the actual failure detection mechanism is supposed to be powerful enough.

3 The basic protocol

The protocol implements a *distributed mutable pointer*, i.e., a pointer that is shared among several threads (on several sites) and whose value can be atomically changed. At any instant, a local representative of the mutable pointer exists on exactly one site; the other sites reference it. We call that representative the *state pointer*. As we said in the introduction, the state pointer is *mobile*, which means that it moves from site to site, so that pointer updates are always performed locally.

3.1 Access structure and thread interface

The distribution in the Mozart platform follows a general scheme [HVS97, HVBS98]. Each site has a node called a *proxy*, and one special site has a node called the *manager*; those nodes are called the *access structure* of the mutable pointer. Any other node accesses the pointer via its local proxy. The protocol manages the movements of the state pointer from proxy to proxy, and updates are done at proxies.

The *thread interface* defines the way threads interact with the protocol. Suppose thread T wants to exchange the mutable pointer's value with *new*. Here is what happens.

1. Thread T sends the message "request(T, new)" to the proxy P on its site.
2. Proxy P can repond in two ways. If P succeeds in the exchange, it returns the old value *old* by sending the message "proceed(old)" to T. If P cannot perform the operation because of

a failure, it replies the message "fail(F)", where the value F describes the failure. The latter message only appears in the extended protocol (Section 5).

This language interface is the foundation of a reflective failure model that permits many useful fault-tolerant abstractions to be realized within the Oz language [VHB99].

3.2 Protocol without path redundancy

The basic, non-fault-tolerant mobile-state protocol is the subject of an earlier paper [VHB+97].

Here is how it works. When a proxy P needs the state pointer to do an exchange, it sends the message "get(P)" to its manager M. The manager then sends the message "forward(P)" to the proxy that will eventually get the state pointer, say P'. Finally proxy P', after having received and updated the state pointer, sends it to P via the message "put(*content*)". At this point, proxy P can exchange the pointer's value locally. Note that the manager only retains the last proxy that will get the state pointer, and it serializes the requests, guaranteeing that there is no starvation.

The protocol has an important property, whose intuition is given here. Suppose proxy P_1 has the state pointer and M receives messages "get(P_2)", "get(P_3)", ... , "get(P_n)", in that order. According to the protocol description, the state pointer will follow the *path* P_1, P_2, P_3, ... , P_n through proxies. We can state that essential result in the following manner.

Theorem 1 (Chain property). *At any moment in time, there exists in the system an ordered sequence of proxies that the state pointer will eventually traverse. This sequence is called the* chain.

Formalizing the chain is the key idea used to prove the protocol correctness [VHB+97].

3.3 Protocol with path redundancy

Now we present an improvement of the basic protocol described above. No fault tolerance is introduced in the protocol, we simply add some *redundancy* in the state of the system (manager and proxies). Later sections show how to use that redundancy to improve the protocol's robustness.

This protocol extension lets the manager maintain a conservative approximation of the *chain* of Theorem 1. We call it the *manager chain* and it has the form

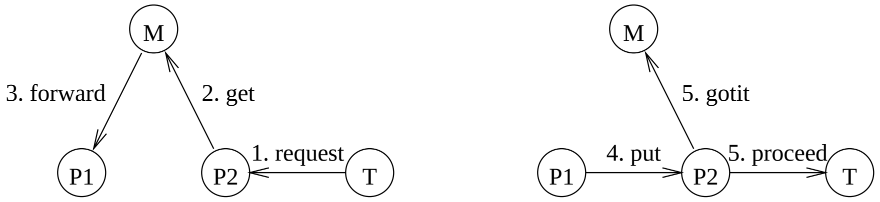
$$[\langle P_1, r_1\rangle, \langle P_2, r_2\rangle, \dots, \langle P_n, r_n\rangle],$$

where $P_1, \dots, P_n$ are proxies and $r_1, \dots, r_n$ are values. Each time a proxy P_i sends a request to its manager, it differentiates that request from its own past requests with a *request identifier* r_i. The reason for using those request identifiers will be explained later. In order that the manager chain contains the needed *path redundancy*, we require the following property to be satisfied.

Requirement 5 (Chain approximation). If the *manager chain* equals $[\langle P_1, r_1\rangle, \dots, \langle P_n, r_n\rangle]$, then there exists m such that $1 \le m \le n$ and the *chain* equals $[P_m, \dots, P_n]$.

The protocol works as follows. When a proxy P needs the state pointer, it sends the message "get(P, r)" to its manager M, where r identifies P's request. The manager receives the message, appends the pair $\langle P, r\rangle$ to its chain and sends the message "forward(P)" to the preceding proxy in the chain, so that the latter forwards the state pointer to P.

When proxy P receives the message "put(*content*)", it sends a new message, "gotit(P, r)", to the manager. The manager receives that message and removes from its chain all the elements before the pair $\langle P, r\rangle$. In this way, the manager chain always contains the proxy that has the state pointer and all the ones that requested it after that proxy. The picture below shows the sequence of messages between two proxies and their manager.



3.4 The nodes' own state

We now describe the participants in the protocol. We consider that the system contains the thread nodes $T_1, \ldots, T_{nt}$, the proxy nodes $P_1, \ldots, P_{np}$, and the manager node M. In Table 1, we define the attributes for each kind of node, with their type and initial value. The attribute *attr* of a node N is noted $N.attr$.

Attribute	*Type*	*Initial value*
$T.id$	NodeRef	ThreadRef(T)
$M.id$	NodeRef	ManagerRef(M)
$M.chain$	(NodeRef $\times$ Identifier)$^+$	$[\langle P_1.id, P_1.requestid\rangle]$
$M.current$	NodeRef $\times$ Identifier	$\langle P_1.id, P_1.requestid\rangle$
$M.knowledge$	$\{\mathrm{OK}, \mathrm{INQ}, \mathrm{SL}\}$	OK
$P.id$	NodeRef	ProxyRef(P)
$P.manager$	NodeRef	ManagerRef(M)
$P.state$	$\{\mathrm{FREE}, \mathrm{CHAIN}\}$	CHAIN if $P = P_1$, FREE otherwise
$P.content$	NodeRef $\cup$ {NULL}	N if $P = P_1$, NULL otherwise
$P.forward$	NodeRef $\cup$ {NULL}	NULL
$P.thread$	NodeRef $\cup$ {NULL}	NULL
$P.newcontent$	NodeRef $\cup$ {NULL}	NULL
$P.requestid$	Identifier	**newId**()
$P.knowledge$	$\{\mathrm{OK}, \mathrm{MF}, \mathrm{SL}\}$	OK

Table 1: Definition of the attributes of thread T, manager M and proxy P

Threads send requests to proxies to exchange the value of the state pointer. All we need here is a node reference for each thread, each proxy and the manager.

The manager maintains an approximation of the *chain* in $M.chain$. Each element of the chain has the form $\langle P, r\rangle$, where P is a reference to a proxy and r is an identifier for the request. This identifier is necessary because a proxy can appear two times in $M.chain$. $M.current$ is an element of $M.chain$ and $M.knowledge$ reflects the manager's knowledge about the fault state of the cell. The latter attributes are used by the inquiry protocol (Section 7).

Proxies receive requests from threads, bring the state pointer on their site, perform the exchange on it, and reply to the thread. When P requested the state pointer, $P.state$ is CHAIN, $P.thread$ refers to the requesting thread, $P.newcontent$ is the new value of the cell, and $P.requestid$ is the current request identifier. When P has the state pointer, $P.state$ is CHAIN, $P.content$ contains the state pointer, and $P.requestid$ is the current request identifier. When P does not have the state pointer and did not request it, *P.state* is FREE, and *P.requestid* is the next request identifier. In all cases, *P.forward* refers to the proxy that P will forward the state pointer to.

The attribute *P.knowledge* reflects P's knowledge about the state of the access structure. It has three possible values: MF if a manager failure has been detected, SL when P knows that the state pointer is lost, and OK otherwise.

State diagrams for proxy, manager and thread are given in Appendix A.

3.5 Formal definition of the basic protocol

The protocol is defined by a set of condition-action rules, that are executed by the manager and the proxies. Rules labelled with a "P" have to be executed on proxies, while the ones with an "M" are intended for manager.

Some auxiliary functions are used in the rules. The pseudo-function **newId**() returns an identifier that is different at every call. The following operations are used on lists. If we assume that $\mathcal{C} = [E_1, E_2, \ldots, E_n]$, then

$$\begin{array}{rlrl}
\mathbf{first}(\mathcal{C}) = E_1 & & E \in \mathcal{C} \equiv \exists i : E = E_i & \\
\mathbf{last}(\mathcal{C}) = E_n & & \mathbf{append}(\mathcal{C}, E) \equiv \mathcal{C} \leftarrow [E_1, E_2, \ldots, E_n, E] & \\
\mathbf{succ}(\mathcal{C}, E_i) = E_{i+1} & & \mathbf{erase}(\mathcal{C}, E_i) \equiv \mathcal{C} \leftarrow [E_1, E_2, \ldots, E_{i-1}, E_{i+1}, \ldots, E_n] & \\
\mathbf{pred}(\mathcal{C}, E_i) = E_{i-1} & & \mathbf{erasebefore}(\mathcal{C}, E_i) \equiv \mathcal{C} \leftarrow [E_i, E_{i+1}, \ldots, E_n] &
\end{array}$$

The rules below are easy to understand. Most of them have the form: "if we receive that kind of message and that condition is true, then we atomically perform these actions." Every rule has a condition which begins with the predicate **at**().

P.1 Request state pointer and reply to thread

$$\begin{array}{l}
\mathbf{at}(P) \wedge \mathbf{receive}(P : \text{request}(T, \mathit{new})) \wedge P.\mathit{content} \neq \text{NULL} \\
\hline
\mathbf{send}(T : \text{proceed}(P.\mathit{content})) \\
P.\mathit{content} \leftarrow \mathit{new}
\end{array}$$

P.2 (a) Request state pointer

$$\begin{array}{l}
\mathbf{at}(P) \wedge \mathbf{receive}(P : \text{request}(T, \mathit{new})) \wedge P.\mathit{state} = \text{FREE} \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\, P.\mathit{knowledge} = \text{OK} \\
\hline
\mathbf{send}(P.\mathit{manager} : \text{get}(P.\mathit{id}, P.\mathit{requestid})) \\
P.\mathit{state} \leftarrow \text{CHAIN} \\
P.\mathit{thread} \leftarrow T \\
P.\mathit{newcontent} \leftarrow \mathit{new}
\end{array}$$

P.3 (a) Accept state pointer

$$\begin{array}{l}
\mathbf{at}(P) \wedge \mathbf{receive}(P : \text{put}(\mathit{old})) \wedge P.\mathit{knowledge} = \text{OK} \\
\hline
\mathbf{send}(P.\mathit{manager} : \text{gotit}(P.\mathit{id}, P.\mathit{requestid})) \\
P.\mathit{content} \leftarrow \mathit{old} \\
\mathbf{if}\ P.\mathit{thread} \neq \text{NULL}\ \mathbf{then} \\
\qquad \mathbf{send}(P.\mathit{thread} : \text{proceed}(P.\mathit{content})) \\
\qquad P.\mathit{thread} \leftarrow \text{NULL} \\
\qquad P.\mathit{content} \leftarrow P.\mathit{newcontent} \\
\mathbf{fi}
\end{array}$$

P.4 Accept forward

$$\begin{array}{l}
\mathbf{at}(P) \wedge \mathbf{receive}(P : \text{forward}(P')) \\
\hline
P.\mathit{forward} \leftarrow P'
\end{array}$$

P.5 (a) Forward state pointer

$\mathbf{at}(P) \wedge P.content \neq \mathrm{NULL} \wedge P.forward \neq \mathrm{NULL} \wedge P.knowledge = \mathrm{OK} \wedge \neg\mathbf{sitedown}(P.forward)$

$\mathbf{send}(P.forward : \mathrm{put}(P.content))$
$P.state \leftarrow \mathrm{FREE}$
$P.content \leftarrow \mathrm{NULL}$
$P.forward \leftarrow \mathrm{NULL}$
$P.requestid \leftarrow \mathbf{newId}()$

M.1 (a) Serialize request

$\mathbf{at}(M) \wedge \mathbf{receive}(M : \mathrm{get}(P, r)) \wedge M.knowledge \neq \mathrm{SL}$

$\langle P', r' \rangle \leftarrow \mathbf{last}(M.chain)$
$\mathbf{send}(P' : \mathrm{forward}(P))$
$\mathbf{append}(M.chain, \langle P, r \rangle)$

M.2 Receive gotit

$\mathbf{at}(M) \wedge \mathbf{receive}(M : \mathrm{gotit}(P, r)) \wedge M.knowledge = \mathrm{OK}$

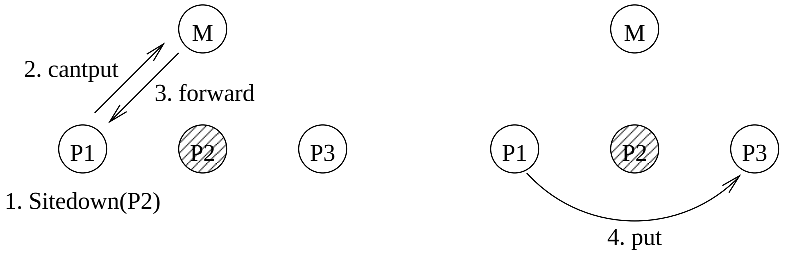
---

if $\langle P, r \rangle \in M.chain$ **then**
    $\mathbf{erasebefore}(M.chain, \langle P, r \rangle)$
    $M.current \leftarrow \langle P, r \rangle$
fi

4 Bypassing a failed proxy

4.1 Description

The second improvement consists in checking whether a proxy is alive before forwarding the state pointer to that proxy. Suppose proxy P_1 has to forward the state pointer to proxy P_2 (see the picture below). If P_1 detects that P_2 has failed, it does not send the state pointer to P_2. Instead P_1 sends the message "cantput(P_1, r_1)" to its manager, where r_1 identifies P_1's last request. To which the manager's response is to send another forward message to P_1 to bypass the failed proxy. So if P_3 is the successor of P_2 in the chain, the manager sends the message "forward(P_3)" to P_1. This means that the state pointer is not affected if sites that do not possess it crash.



It is possible to extend the bypass process to the case of unreachable proxies [BVCK00].

4.2 Formal definition

The basic protocol is extended by adding condition-action rules to the ones given in section 3.5. Compare rules P.5(b) and P.5(c) with P.5(a), and P.2(b) with P.2(a).

P.5 (b) Bypass failed proxy

$\mathbf{at}(P) \wedge P.content \neq \mathrm{NULL} \wedge P.forward \neq \mathrm{NULL} \wedge P.knowledge = \mathrm{OK} \wedge \mathbf{sitedown}(P.forward)$

$\mathbf{send}(P.manager : \mathrm{cantput}(P.id, P.requestid))$
$P.forward \leftarrow \mathrm{NULL}$

M.3 Receive cantput

$$\frac{\mathbf{at}(M) \wedge \mathbf{receive}(M : \text{cantput}(P, r))}{\begin{array}{l} \langle P', r' \rangle \leftarrow \mathbf{succ}(M.chain, \langle P, r \rangle) \\ \mathbf{erase}(M.chain, \langle P', r' \rangle) \\ \textbf{if } \langle P, r \rangle \neq \mathbf{last}(M.chain) \textbf{ then} \\ \quad \langle P'', r'' \rangle \leftarrow \mathbf{succ}(M.chain, \langle P, r \rangle) \\ \quad \mathbf{send}(P : \text{forward}(P'')) \\ \textbf{fi} \end{array}}$$

5 Proxy behavior in case of manager failure

5.1 Description

The third improvement of the protocol is to allow proxies to detect a manager failure. As we saw above, the manager is a key entity in the protocol. When the manager fails, almost everything is "freezed". The proxy that has the state pointer will keep it forever and continue to do exchanges. The other ones will eventually detect the manager failure. From that on, they know they will never get the state pointer.

The reader may be concerned about the dependency on the manager site. This strong dependency is a consequence of the protocol's efficiency, i.e., that it has the same message latency as the non-fault-tolerant protocol when there is no failure. Adding a small amount of centralized redundancy (the chain in the manager) can give reasonable fault-tolerant behavior without sacrificing performance. So the dependency should be judged in the context of the complete system. This protocol is a basic building block, which provides efficient means to build more robust fault-tolerance within the Oz language, if needed.

5.2 Formal definition

The predicate $\mathbf{error}(P)$ describes P's knowledge about the error state of the system. Its possible values are *ok* (no known problem), *sl/foreign* (manager down, pointer not at P), *sl/home* (manager down, pointer at P), *sl* (state lost everywhere). Those values can be deduced from P's attributes.

Rule P.3(b) is the complement of rule P.3(a). It shows that if during the time the state pointer is in the network, the destination proxy detects manager failure, then it is completely lost. The rules P.6 implement the missing part of the thread interface.

P.3 (b) Drop state pointer

$$\frac{\mathbf{at}(P) \wedge \mathbf{receive}(P : \text{put}(old)) \wedge P.knowledge \neq \text{OK}}{\mathbf{skip}}$$

P.6 (a) Report error

$$\frac{\mathbf{at}(P) \wedge P.thread \neq \text{NULL} \wedge \mathbf{error}(P) \neq ok}{\begin{array}{l} \mathbf{send}(P.thread : \text{fail}(\mathbf{error}(P))) \\ P.thread \leftarrow \text{NULL} \end{array}}$$

P.6 (b) Report error

$$\frac{\begin{array}{r} \mathbf{at}(P) \wedge \mathbf{receive}(P : \text{request}(T, new)) \wedge P.content = \text{NULL} \\ \wedge\, \mathbf{error}(P) \neq ok \end{array}}{\mathbf{send}(T : \text{fail}(\mathbf{error}(P)))}$$

P.7 (a) Detect manager failure

$$\frac{\mathbf{at}(P) \wedge P.knowledge = \text{OK} \wedge \mathbf{sitedown}(P.manager)}{P.knowledge \leftarrow \text{MF}}$$

6 Proving correctness

To prove the protocol correct we proceed in three steps. First, we propose a protocol invariant, that is, a succinct characterization of the system state at any moment in time. We call it the *chain invariant CI* and an intuition of it is given below. A complete specification of the invariant is given in [BVCK00].

Consider the set of proxies in the system, i.e., $\{P_1, \ldots, P_{np}\}$. The chain invariant states that it can be partitioned among four sets A, B, C and D. The four sets have the following characteristics:

- The proxies in A form the *chain* of Theorem 1. Each proxy P in A has a reference to the next one in the chain. That reference is either in *P.forward*, or in a "forward" message addressed to P, or it can even be lost if the manager has failed.
- The proxies in B have sent a "get" message to the manager, which did not receive it yet.
- The proxies in C are free, they have not requested the state pointer.
- The proxies in D are failed ones.

The chain invariant also states the *path redundancy*, i.e., Requirement 5. One easily sees that the protocol makes the proxies move from one set to an other.

The next step is to prove that this statement is actually invariant. Finally, we prove that the protocol does what it is supposed to do, namely correctly move and update the cell's content. The statements of those results are given below, details can be found in [BVCK00].

Theorem 2 (Safety). *The chain invariant CI is an invariant of the mobile-state protocol.*

Theorem 3 (Liveness). *Assume that the execution of all the rules is strongly fair. If the state pointer is requested at proxy P, then exactly one of the following three statements holds.*

1. *The manager site does not fail and the state pointer is never lost. Then P will eventually receive the state pointer exactly once.*
2. *The manager site does not fail and the state pointer is lost before it reaches P. Then P will never receive the state pointer.*
3. *The manager site fails, and the proxy is notified of this. If it does not have the state pointer, it knows that it will never receive it.*

7 When the state is lost: the inquiry protocol

7.1 Description

The last improvement of the protocol is to add an inquiry protocol that permits to detect when the state pointer is definitely lost. This loss can happen in two ways:

Case 1. The state pointer is at proxy P and P's site crashes.

Case 2. The state pointer has been sent over the network in a "put" message and the message is lost because of a site failure (of the sender or the receiver).

The procedure works as follows. The manager sends the message "inquire(P', r)" to a proxy P in its chain, where P' is P's predecessor and r is P's request identifier. The proxy can respond "inform($AFTERME$)" (the proxy already received the state pointer and it forwarded it), "inform($ATME$)" (the proxy has the state pointer), "inform($BEFOREME$)" (the proxy did not received the state pointer yet) or "inform($CANTREC$)" (the proxy detected that P' is failed). The manager then uses that response to pinpoint where the state pointer is, and possibly to notify other proxies of a state loss. Note that the inquiry protocol, which is relatively expensive, is invoked only if there is a failure.

There is also an invariant property with the inquire protocol, and it is similar to the preceding one. And there is a liveness theorem. The new liveness theorem extends Theorem 3 as follows: statement 2 is replaced by

2'. The manager site does not fail and the state pointer is lost before it reaches P. Then P will never receive the state pointer, but it will eventually receive notification from the manager that the state pointer is lost.

All the details can be found in [BVCK00].

8 Conclusions

We have defined a fault-tolerant mobile-state protocol for the Mozart platform, which implements the Oz language. The protocol with inquiry is implemented at the heart of Mozart's concurrent object system. When there is no failure, this protocol has the same network overhead as its non-fault-tolerant ancestor [VHB+97]. When there are failures, the protocol provides sufficient hooks to the requesting threads to allow common fault-tolerant algorithms to be implemented completely within the Oz language (an example is given in [AV00]). So the protocol never makes irrevocable decisions. We prove in [BVCK00] that the protocol correctly updates object state as long as the state itself is not lost and one other site (the manager site) does not crash.

Acknowledgements

This research is partly financed by the Walloon Region of Belgium.

References

[AV00] Iliès Alouini and Peter Van Roy. An open distributed fault-tolerant transactional store in Mozart. In preparation, 2000.

[BVCK00] Per Brand, Peter Van Roy, Raphaël Collet, and Erik Klintskog. A primitive mobile-state protocol for constructing fault-tolerant abstractions. Technical report, Université catholique de Louvain, 2000. In preparation, draft available at `http://www.info.ucl.ac.be/~raph/research`.

[DH98] M. J. Demmer and M. P. Herlihy. The arrow distributed directory protocol. *Lecture Notes in Computer Science*, 1499:119–133, 1998.

[Gär99] Felix C. Gärtner. Fundamentals of fault tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[HF99] Seif Haridi and Nils Franzén. Tutorial of Oz. Technical report, 1999. In Mozart documentation, available at `http://www.mozart-oz.org`.

[HVBS98] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3), May 1998.

[HVS97] Seif Haridi, Peter Van Roy, and Gert Smolka. An overview of the design of Distributed Oz. In the *2nd International Symposium on Parallel Symbolic Computation* (*PASCO 97*), New York, July 1997. ACM.

[Moz99] Mozart Consortium (DFKI, SICS, UCL, UdS). The Mozart programming system (Oz 3), January 1999. Available at `http://www.mozart-oz.org`.

[OMG99] Object Management Group. The common object request broker: Architecture and specification. Technical report, OMG, June 1999. Rev. 2.3.

[Sun98] Java remote method invocation specification. Technical report, Sun Microsystems, 1998.

[VHB+97] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.

[VHB99] Peter Van Roy, Seif Haridi, and Per Brand. Distributed programming in Mozart – A tutorial introduction. Technical report, 1999. In Mozart documentation, available at `http://www.mozart-oz.org`.

[WWWK94] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories and Mountain View, CA, November 1994.

A State diagrams

The picture below gives the state diagrams for proxy, manager and thread nodes in the full protocol. States are given in circles. In order to increase readability, error states are placed on the right of the diagrams and some states are grouped together into macro-states. Arrows represent possible state transitions. They are labelled with pairs *consumed event/generated event*, where events are messages or failure detection. These diagrams show four of the refinements as well.