A Lightweight Reliable Object Migration Protocol

Peter Van Roy[1,2], Per Brand[2], Seif Haridi[2], and Raphaël Collet[1]

[1] Université catholique de Louvain,
B-1348 Louvain-la-Neuve, Belgium
{pvr,raph}@info.ucl.ac.be
http://www.info.ucl.ac.be

[2] Swedish Institute of Computer Science,
S-164 28 Kista, Sweden
{perbrand,seif}@sics.se
http://www.sics.se

Abstract. This paper presents a lightweight reliable object migration protocol that preserves the centralized object semantics, allows for precise prediction of network behavior, and permits construction of fault tolerance abstractions in the language. Each object has a "home site" to which all migration requests are directed. Compared to the standard technique of creating and collapsing forwarding chains, this gives a better worst-case network behavior and it limits dependencies on third-party sites. The protocol defines "freely mobile" objects that have the interesting property of always executing locally, i.e., each method executes in the thread that invokes it. This makes them dual, in a precise sense, to stationary objects. The protocol is designed to be as efficient as a nonreliable protocol in the common case of no failure, and to provide sufficient hooks so that common fault tolerance algorithms can be programmed completely in the Oz language. The protocol is fully implemented in the network layer of the Mozart platform for distributed application development, which implements Oz (see `http://www.mozart-oz.org`). This paper defines the protocol in an intuitive yet precise way using the concept of *distribution graph* to model distributed execution of language entities. Formalization and proof of protocol properties are done elsewhere.

1 Introduction

What does it mean for an object to be mobile? Different systems interpret this notion in different ways [15]:

1. Only code is copied, not the state (e.g., Java applets [13]).
2. The original object is "frozen" and its code and state are both copied. The original is discarded before the copy is used. This is what is typically meant by "mobile agents" in programming books.
3. Concurrency control is added by freezing the original object, copying both its code and state, making it forward all messages, and then unfreezing it.

The copy maintains lexical scoping across the network by using network references for all of the original object's external references. This approach is taken by Emerald [10] and Obliq [5].

If the application programmer wants to control network behavior by placing objects on the appropriate sites and moving them as desired, then all three approaches have problems:

Bad network behavior If an alias is created from the old object to the new when it moves, then a sequence of moves creates a chain of aliases. There are various tricks to reduce the length of this chain, e.g., Emerald short-circuits it when a message is sent, but a basic unpredictability remains regarding third-party dependencies and number of network hops. Furthermore, if there is a site failure, Emerald uses a broadcast to recover the object, which is impractical on a WAN.

Weak semantics One way to get better network behavior is give up on the transparent semantics, i.e., to use one of the two first approaches above.

Is it possible to get both a predictable network behavior and a good semantics? A simple way to solve this problem is to make objects mobile by default, instead of stationary. We call these "freely mobile objects". Start with lightweight object mobility, in which a method always executes in the thread that invokes it. Implement this with a reliable distributed algorithm that guarantees predictable network behavior. Then control the mobility by *restricting* it. In this way, we achieve arbitrary mobility while keeping both predictable network behavior and good semantics. One purpose of this paper is to present a distributed algorithm that achieves these goals.

This paper consists of four parts. Section 2 introduces the graph model of distributed execution and its visual notation. Section 3 defines a home-based algorithm for freely mobile objects and discusses its properties and usefulness. Section 4 defines the basic mobile state protocol, which is the key algorithm at the heart of freely mobile objects [4, 19]. Section 5 extends the basic protocol to use precise failure detection, while maintaining the same performance as the basic protocol when there is no failure. This section also gives the state diagrams of both the basic and extended protocols. The extended protocol is part of the implementation of the Mozart programming system [11, 9], which implements the Oz language. Oz is a multiparadigm language with simple formal semantics that can be viewed as a concurrent object-oriented language with dataflow synchronization [8].

The extended protocol is being used to implement nontrivial fault tolerance abstractions in Oz [18, 17]. For example, we are currently testing an open distributed fault-tolerant transactional store [3, 7]. Clients to this store can come and go, and the store remains coherent as long as at least one working client exists at any given moment. A fuller explanation of the design of fault tolerance abstractions is beyond the scope of this paper.

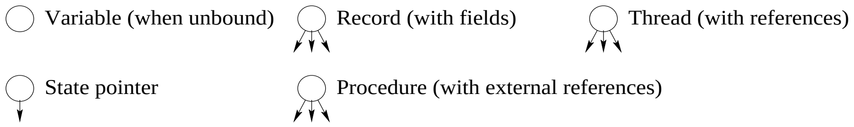

Fig. 1. Language entities as nodes in a graph

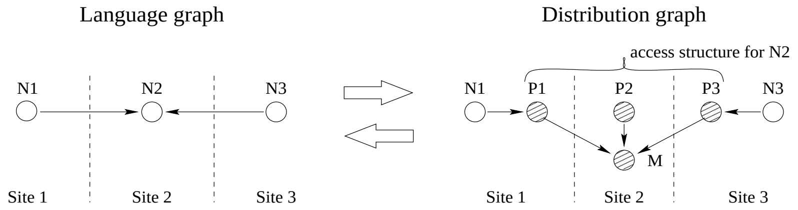

Fig. 2. Access structure in the distribution graph

2 Graph Notation for Distributed Executions

We depict distributed executions in a simple but precise manner using the concept of *distribution graph*. This section gives a self-contained explanation that suffices to understand the paper. We obtain the distribution graph in two steps from an arbitrary execution state of the system. The first step is independent of distribution. We model the execution state by a graph, called *language graph*, in which each language entity except for an object corresponds to one node (see Figure 1). A node can be considered as an active entity with internal state that can asynchronously send messages to other nodes and that can receive messages from other nodes. In terms of the language graph, Oz objects are compound entities, i.e., they consist of more than one node. Their structure is explained in Section 3.

In the second step, we introduce the notion of *site*. Assume a finite set of sites and annotate each node by its site (see Figure 2). If a node, e.g., `N2`, is referenced by at least one node on another site, then map it to a *set* of nodes, e.g., {`P1`,`P2`,`P3`,`M`}. This set is called the *access structure* of the original node. The map satisfies the following invariant. An access structure consists of at most one *proxy node* `Pi` on each site that referenced the original node and exactly one *manager node* `M` for the whole structure. The manager site is also called the *home site*. The resulting graph, containing both local nodes and access structures where necessary, is called the *distribution graph*. All example executions in the paper use this notation.

All distributed execution is modeled as distribution graph transformations. Nodes in the distribution graph are active entities with internal state that can send and receive messages. All graph transformations are atomic and initiated by the nodes. The two-level addressing mechanism by which a node identifies a send destination is beyond the scope of this paper (see [2]). We mention only that each access structure has a globally unique name, which is used to identify its nodes.

```
class Account
   attr bal:0
   meth trans(Amt)
      bal<- @bal+Amt
   end
   meth getBal(B)
      B = @bal
   end
end

A={New Account trans(100)}
```

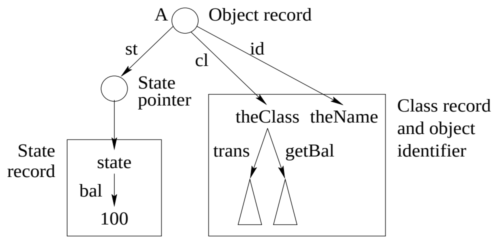

Fig. 3. An object with one attribute and two methods

3 Freely Mobile Objects

In the distribution graph, an object shows up as a compound entity consisting of an object record, a class record containing procedures (the methods), a state pointer, and a record containing the object's state. The distributed behavior of the object is derived from the behavior of its parts. Figure 3 shows an object `A` that has one attribute, `bal`, and two methods, `trans` and `getBal`. The object is represented as an object record with three fields. The `st` field contains a state pointer, which points to the object's state record. The state pointer defines the site at which state updates can be done without network operations. The `cl` field contains the class record, which contains the procedures `trans` and `getBal` that implement the methods. The `id` field contains the object's unique identifier `theName`. The object record and the class record cannot be changed. However, by giving a new content to the state pointer, the object's state can be updated.

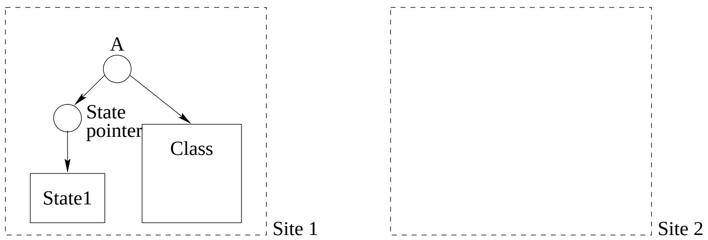

Fig. 4. A local object

Figure 4 shows an object `A` that is local to Site 1. There is no reference to `A` from any other site. Figure 5 shows an object `A` with one remote reference. The

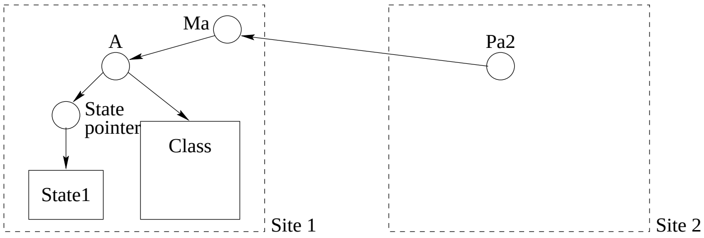

Fig. 5. A global object with one remote reference

object is now part of an access structure whose manager is on Site 1 and that has one proxy on Site 2. A local object `A` is transformed to a global (i.e., remotely-referenced) object when a message referencing `A` leaves Site 1. A manager node `Ma` is created on Site 1 when the message leaves. When the message arrives on Site 2, then a proxy node `Pa2` is created there.

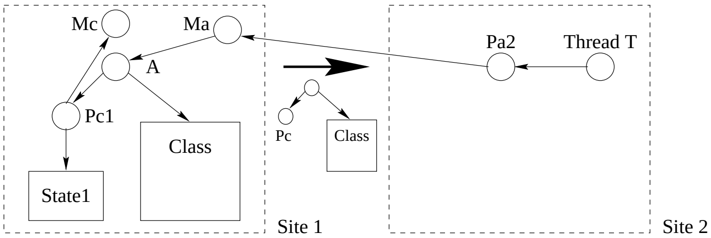

Fig. 6. The object is invoked remotely (1)

Figure 6 shows what happens when thread `T` invokes `A` from Site 2. At first, only the proxy `Pa2` is present on Site 2, not the object itself. The proxy asks its manager for a copy of the object record. This causes an access structure to be created for the state pointer, with a manager `Mc` and one proxy `Pc1`. The class record is copied eagerly and does not have a global name. A message containing the class record and a state pointer proxy is sent to Site 2. The object's state remains on Site 1.

Figure 7 shows what happens when the message arrives. A second proxy `Pc2` is created for the state pointer. The class record is copied to Site 2 and proxy `Pa2` becomes the object record `A`. The mobile state protocol (see Section 4) then atomically transfers the state pointer to Site 2. The object record has a global name. This implies that any further messages to Site 2 containing object references will immediately refer to the local copy of the object record. No additional network operations are needed.

Figure 8 shows what happens after the state pointer is transferred to Site 2. The new state, `State2`, is created on Site 2 and will contain the updated object

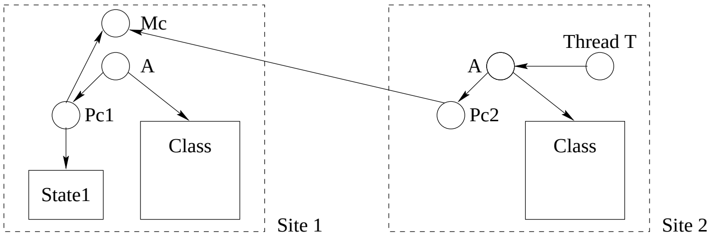

Fig. 7. The object is invoked remotely (2)

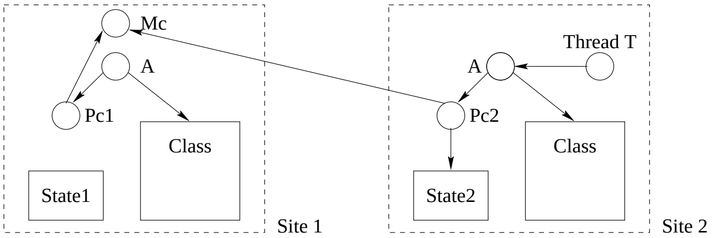

Fig. 8. The object is invoked remotely (3)

state after the method finishes. The old state, `State1`, may continue to exist on Site 1 but the state pointer no longer points to it.

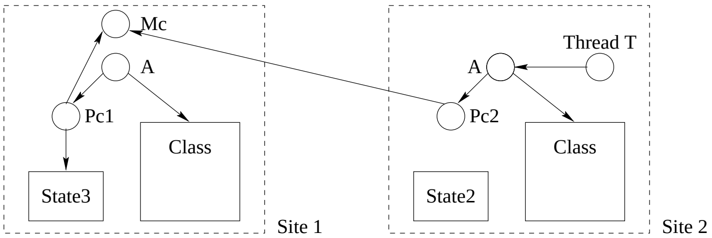

Fig. 9. The object moves back to Site 1

Figure 9 shows what happens if Site 1 invokes the object again. The state pointer is transferred back to Site 1. The new state, `State3`, is created on Site 1 and will contain the updated object state after the method finishes. The old state, `State2`, may continue to exist on Site 2 but the state pointer no longer points to it.

3.1 Discussion

There are several interesting things going on here. First, the object is always executed locally. The state pointer is always localized before the method starts executing, and it is guaranteed to stay local during the method execution while the object is locked. Second, the class code is only transferred once to any site. Only the state pointer is moved around after the first transfer. This makes object mobility very lightweight. Third, all requests for the object are serialized by the state pointer's manager node. This simplifies the protocol but introduces a dependency on the manager site (i.e., the protocol is "home-based" [12]). This dependency can be removed at higher levels of abstraction (e.g., see [3], which is also a migration protocol) or by extending the present protocol (see [16] for a simple solution).

A stationary object can be defined in a system in which the only mobile entities are freely mobile objects. This requires remote thread creation, synchronization between different sites, and passing of exceptions between sites. An Oz procedure that takes any freely mobile object and returns a reference to a stationary object with the same language semantics is given in [19]. The procedure's definition makes it clear that a stationary object is not a simple concept. Freely mobile objects are simpler since they always execute in the thread that invokes them.

There is a precise sense in which stationary and freely mobile objects are duals of each other. With a stationary object, the object state remains on one site, and the thread conceptually moves to that site, to execute there. With a freely mobile object, it is the reverse: the thread remains on one site, and the object state moves to that site. Since moving object state is simpler than moving a thread (see, e.g., [10, 14]), this confirms that freely mobile objects are simpler than stationary objects.

4 The Basic Mobile State Protocol

The mobile objects of Section 3 are compound entities that use several distributed algorithms. The object record is copied once lazily (when the object is first invoked), the methods are copied along with it, and the object's state pointer is moved between sites that request it. The protocol that moves the state pointer, the *mobile state* protocol, is particularly interesting because of the way it is integrated into the object system. In this section we give the basic protocol that assumes there is no failure. Section 5 explains how the protocol is extended with failure detection.

The protocol must guarantee consistency between consecutive states. If the consecutive states are on different sites, then this requires an atomic transfer of the state pointer between the sites. A site that wants the state pointer requests it from the state pointer's manager, and the latter sends a forwarding command to the site that has (or will eventually have[1]) the state pointer. Therefore, in the basic protocol, the manager only needs to store one piece of information, namely the site that will eventually contain the state pointer [19].

We show how the protocol works by means of an example. Figure 10 shows a state pointer `C` referenced from two sites. The state pointer is initially on Site

[1] In the temporal logic sense [1].

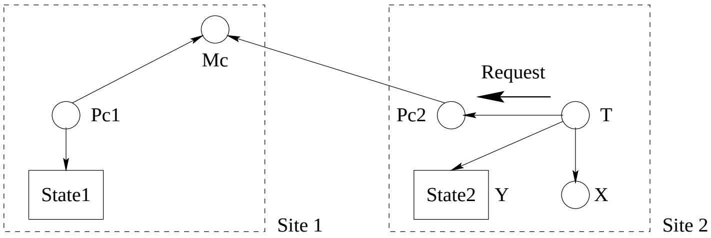

Fig. 10. The state pointer is referenced by Pc1; thread T requests a state update

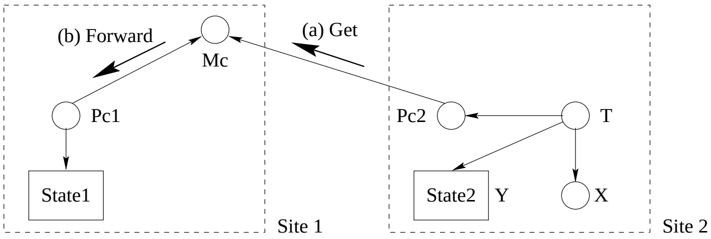

Fig. 11. (a) Pc2 requests the state pointer; (b) Pc1 is asked to forward it

1; proxy `Pc1` has the state pointer and proxy `Pc2` does not. Thread `T` executes on Site 2 an object method that will update the object state. At some point, `T` requests a state update by sending a `Request` message to `Pc2`. The thread references the new state through variable `Y`. When the update completes then the thread will also reference the old state through variable `X`.

Since proxy `Pc2` does not have the state pointer, it must ask its manager. Figure 11 shows (a) `Pc2` requesting the state pointer by sending a `Get` message to manager `Mc`, and (b) the manager sending a `Forward` message to the proxy that will eventually have the state pointer, namely `Pc1`. Therefore the manager can accept another request immediately; it does not need to wait until the state pointer's transfer is complete.

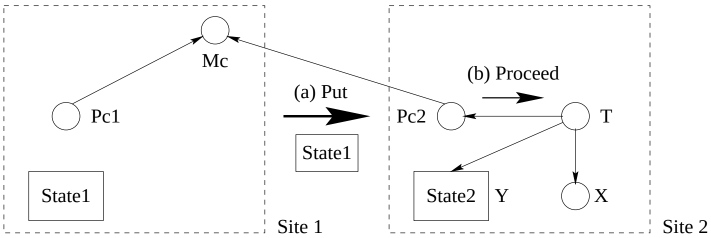

Fig. 12. (a) Pc1 forwards the state pointer; (b) Pc2 informs thread T

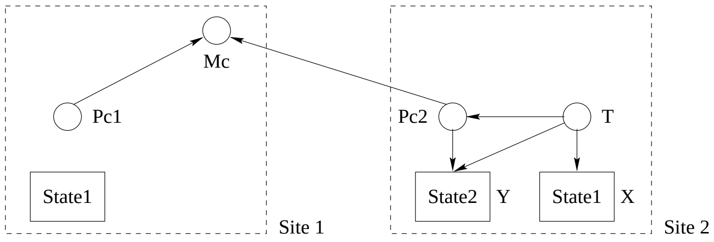

Fig. 13. Pc2 has the state pointer; T references the old and new states

Figure 12 shows (a) `Pc1` sending to `Pc2` a `Put` message containing the old state, `State1`, and (b) `Pc2` sending to `T` a `Proceed` message informing it that the transfer is complete. The old state may still exist on Site 1 but `Pc1` no longer has a pointer to it. Figure 13 shows the final situation. `Pc2` has the state pointer, which points to `State2`, and `X` is bound to `State1`. Therefore `T` references both `State1` and `State2`.

This protocol provides a predictable network behavior. There are a maximum of three network hops for the state pointer to change sites; only two if the manager is on the source or destination site; zero if the state pointer is on the requesting site. The protocol maintains sequential consistency, that is, updates to the state pointer are done in a globally consistent order.

5 The Reliable Mobile State Protocol

We extend the basic protocol to obtain a new protocol that provides reliable failure detection and that is a sufficient foundation for building fault tolerance abstractions. The resulting protocol satisfies the following theorem:

Theorem 1 (Failure detection theorem). *If the state pointer is requested at proxy P, then exactly one of the following three statements is eventually true:*

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 the state pointer 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.*
3. *The manager site fails. Then P is notified of this. If it does not have the state pointer, then it infers that it will never receive it.*

The proof of this theorem is given elsewhere [4]. An important corollary of this theorem is that the protocol has no time outs. Deciding whether or not to time out is left to the application.

Figure 14 gives the essential parts of the thread, proxy, and manager state diagrams for the basic protocol and its reliable extension. Each circle represents

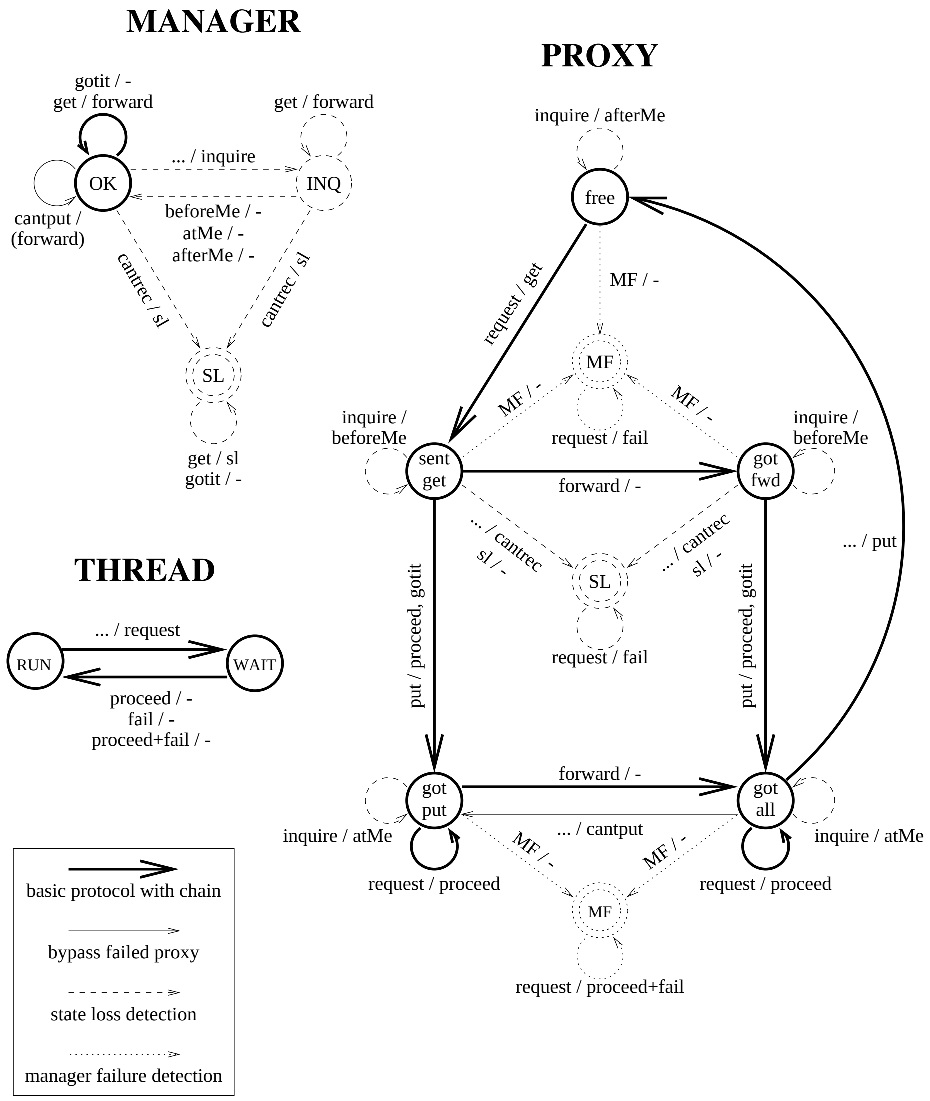

Fig. 14. State diagrams of the basic and reliable protocols

a proxy state or manager state. A transition arrow represents an atomic state change; each arrow is optionally labeled *Condition / Action*. Here *Condition* is the firing condition (often a received message, but possibly a network condition), and *Action* is an action to be performed on firing (often a sent message). MF means "manager failure," SL means "state lost," and "-" means no action. To avoid overloading the figure, message arguments are left out and some conditions are given as "...". In addition, the transitions for network inactivity are not shown. They can be added by assuming that each state has a second "mirror" state. The system transfers to a mirror state if network inactivity is detected. The system transfers back to the original state when the network becomes active again.

In Figure 14, the basic protocol is defined by the thick black lines. The extensions are given in three further steps: bypass failed proxy, state loss detection, and manager failure detection. The rest of this section justifies the extended protocol. First, Section 5.1 defines the failure models of the network and of the state pointer. Then, Section 5.2 constructs the reliable protocol by adding functionality in three steps, as shown in the figure.

5.1 The failure models

The reliable protocol assumes the network failure model and implements the state pointer failure model. The network is assumed to send messages in asynchronous and unordered (i.e., non-FIFO) fashion. Messages that arrive are not corrupted.

The network failure model The network has two failure modes: network inactivity and permanent site failure. In our experience, these two modes cover the vast majority of failures in an application environment using the TCP/IP protocol family on the Internet [6]. We assume site failures are instantaneous and permanent. Messages in transit from a failed site may be lost. Messages to a failed site are always lost. Network inactivity is detected as a sudden decrease in communication bandwidth between a pair of sites. We assume network inactivity is detected quickly, e.g., in much less time than a TCP time out. We also assume the network will eventually become active again. This might or might not happen while the application is executing. If the network becomes active again, then no messages are lost.

The state pointer failure model Given the network failure model, the reliable protocol is designed so that a proxy can inform a calling thread of the following problems:

- Permanent inability to perform a state update (state lost).
- Permanent inability to move the state pointer (manager failure); the state pointer may or may not be local.
- Current inability to perform a state update (network inactivity). This may go away, if the network inactivity goes away.

– Current inability to move the state pointer (network inactivity); the state pointer may or may not be local. This may go away, if the network inactivity goes away.

The Mozart system can be configured at run-time to detect these failures synchronously or asynchronously, and to raise an exception or call a user-defined procedure when a particular failure is encountered on a given state pointer [18].

5.2 Stepwise construction of the reliable protocol

We construct the reliable protocol in stepwise fashion from its nonreliable ancestor, following the steps of Figure 14. We first introduce the concept of *proxy chain*, which at any instant is the sequence of proxy nodes that the state pointer will eventually traverse. If several proxies send `Get` messages in quick succession, then it may take some time before the state pointer has visited them all. The proxy chain represents at any instant the sequence of proxies that are still waiting for the state pointer [19]. In the basic protocol, the proxy chain is stored as a distributed data structure.

Basic protocol with chain The first improvement is to let the manager maintain a conservative approximation to the proxy chain (see Figure 15). This is very simple: when the manager receives a `Get` message, it appends the requesting proxy to the proxy chain. It then sends a `Forward` message to the preceding proxy, so that the latter forwards the state pointer to the requesting proxy. When a proxy receives a `Put` message containing the state pointer, it sends a new message, `Gotit`, to the manager. The `Gotit` message does not exist in the basic protocol. When the manager receives the `Gotit`, then it removes from the proxy chain all proxies before the one that sent the `Gotit`. Moving the state pointer costs four messages instead of three for the basic protocol, but the message latency does not change, i.e., it is still three messages.

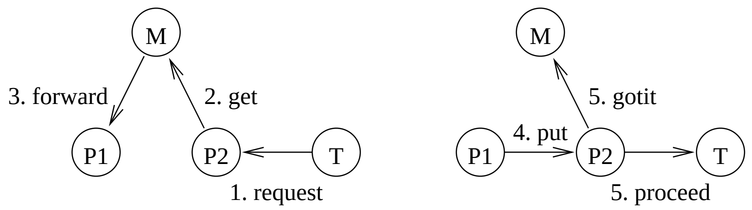

Fig. 15. Basic protocol with chain

Bypass failed proxy The second improvement consists in checking whether a proxy is working before forwarding the state pointer to that proxy (see Figure 16). Suppose proxy P_1 has to forward the content to proxy P_2. If P_1 detects that P_2 has failed, then it sends a new message, `Cantput`, to the manager. The manager then sends another `Forward` message to P_1 to bypass the failed proxy. Therefore the state pointer can survive crashes of sites that do not possess it.

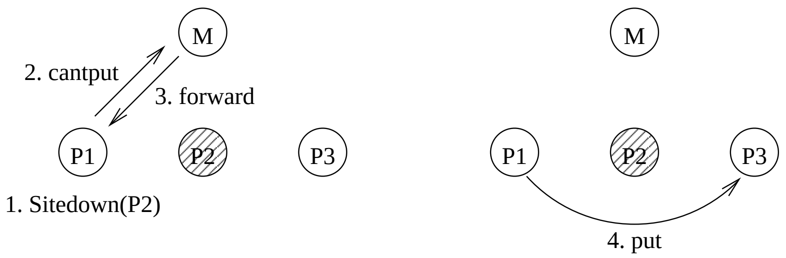

Fig. 16. Bypass failed proxy

State loss detection The third improvement is to add an inquiry protocol that determines when the state pointer is definitely lost. This loss can happen in two ways:

- The state pointer is at proxy P and P's site crashes.
- 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 inquiry protocol is implemented at the manager node. The inquiry protocol traverses the chain and asks each proxy where the state pointer is. The proxy answers with `beforeMe`, `atMe`, or `afterMe`. The basic idea is to bracket the state pointer's location. If the protocol finds two proxies that answer `afterMe` and `beforeMe`, and all proxies between them have crashed, and there is nothing in the network, then the state pointer is lost.

Manager failure detection The fourth and last improvement is to allow proxies to detect a manager failure. This is useful in the following situations:

- Proxy P does not have the state pointer and wishes to obtain it. The proxy infers that it will never receive the state pointer and it can directly signal that fact to the thread.
- Proxy P has the state pointer and cannot forward it. The proxy infers that it will keep the state pointer forever.

6 Conclusions and reflections

This paper presents the main ideas of an efficient, reliable object migration protocol that preserves centralized object semantics and allows for precise prediction of network behavior. The main limitation of the protocol is its dependency on the manager site. This limitation will be removed in the future [16]. The migration protocol is completely implemented as part of Mozart, a platform for distributed application development based on the Oz language [11]. We are using Mozart to write efficient and robust distributed applications (e.g., [7]).

We introduce the concept of "freely mobile object," whose distributed semantics are implemented by the migration protocol. Freely mobile objects are interesting because they are always executed locally, i.e., each method is executed in the thread that invokes it. The same property holds for centralized objects. Stationary objects, which are always executed on the same site, are more complex beasts on any system. For example, Java RMI semantics must define which threads are used for remote calls and when new threads are created [14]. In Oz, one can define stationary objects *in* the language. They require remote thread creation, synchronization between threads on different sites, and passing exceptions between threads on different sites.

We provide evidence that a freely mobile object is a useful basis for a system with migratory objects. Freely mobile objects behave as state caches, and as such provide a mechanism for latency tolerance. Furthermore, arbitrary mobility behavior can be programmed at the language level by restricting the mobility of freely mobile objects. At all times, the language semantics of objects are respected.

Acknowledgements

This research is partially financed by the Walloon Region of Belgium. The reliable migration protocol was designed by Per Brand and the first version of the Mozart network layer was implemented by him. We thank Erik Klintskog for his implementation of fault-tolerant objects in Oz using the protocol presented here. We thank Iliès Alouini for his implementation of a fault-tolerant transactional store in Oz. Finally, we thank all the other contributors and developers of the Mozart Programming System.

References

1. Mack W. Alford, Leslie Lamport, and Geoff P. Mullery. chapter 2. Lecture Notes in Computer Science, vol. 190. Springer Verlag, 1985. Basic Concepts, in Distributed Systems–Methods and Tools for Specification, An Advanced Course.
2. Iliès Alouini and Peter Van Roy. Le protocole réparti de Distributed Oz (The distributed protocol of Distributed Oz) (in French). In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP 99)*, pages 283–298. Hermès Science Publications, April 1999.

3. Iliès Alouini and Peter Van Roy. An open distributed fault-tolerant transactional store in Mozart. In preparation, 2000.
4. Per Brand, Peter Van Roy, Raphaël Collet, and Erik Klintskog. A fault-tolerant mobile-state protocol and its language interface. In preparation, 1999.
5. Luca Cardelli. A language with distributed scope. In *Principles of Programming Languages (POPL)*, pages 286–297, January 1995.
6. Douglas E. Comer. *Internetworking with TCP/IP. Vol. 1: Principles, Protocols, and Architecture.* Prentice-Hall, Englewood Cliffs, N.J., 1995.
7. Donatien Grolaux. Editeur graphique réparti basé sur un modèle transactionnel (A distributed graphic editor based on a transactional model) (in French). Technical report, Université catholique de Louvain, June 1998. Mémoire de fin d'études (Master's project).
8. Seif Haridi and Nils Franzén. Tutorial of Oz. Technical report, 1999. Draft. In Mozart documentation, available at `http://www.mozart-oz.org`.
9. Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3), May 1998.
10. 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.
11. Mozart Consortium (DFKI, SICS, UCL, UdS). The Mozart programming system (Oz 3), January 1999. Available at `http://www.mozart-oz.org`.
12. James D. Solomon. *Mobile IP–The Internet Unplugged.* PTR Prentice Hall, Upper Saddle River, New Jersey, 1998.
13. Sun Microsystems. *The Java Series.* Mountain View, Calif., 1996. Available at `http://www.aw.com/cp/javaseries.html`.
14. Sun Microsystems. *The Remote Method Invocation Specification.* Sun Microsystems, Mountain View, Calif., 1997. Available at `http://www.javasoft.com`.
15. Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997.
16. Peter Van Roy. Eliminating manager dependency in mobile cell protocol, August 1999. Message on internal Mozart mailing list.
17. Peter Van Roy. On the separation of aspects in distributed programming: Application to distribution structure and fault tolerance in Mozart. In *International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA 99)*, Tohoku University, Sendai, Japan, July 1999.
18. Peter Van Roy, Seif Haridi, and Per Brand. Distributed programming in Mozart – A tutorial introduction. Technical report, Mozart Consortium, 1999. Draft. In Mozart documentation, available at `http://www.mozart-oz.org`.
19. 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.