Programming Languages for Distributed Applications

Seif Haridi,[*] Peter Van Roy,[†] Per Brand,[‡] and Christian Schulte[§]

March 23, 1998

Abstract

Much progress has been made in distributed computing in the areas of distribution structure, open computing, fault tolerance, and security. Yet, writing distributed applications remains difficult because the programmer has to manage models of these areas explicitly. A major challenge is to integrate the four models into a coherent development platform. Such a platform should make it possible to cleanly separate an application's functionality from the other four concerns. Concurrent constraint programming, an evolution of concurrent logic programming, has both the expressiveness and the formal foundation needed to attempt this integration. As a first step, we have designed and built a platform that separates an application's functionality from its distribution structure. We have prototyped several collaborative tools with this platform, including a shared graphic editor whose design is presented in detail. The platform efficiently implements Distributed Oz, which extends the Oz language with constructs to express the distribution structure and with basic primitives for open computing, failure detection and handling, and resource control. Oz appears to the programmer as a concurrent object-oriented language with dataflow synchronization. Oz is based on a higher-order, state-aware, concurrent constraint computation model.

1 Introduction

Our society is becoming densely interconnected through computer networks. Transferring information around the world has become trivial. The Internet, built on top of the TCP/IP protocol family, has doubled in number of hosts every year since 1981, giving more than 20 million in 1997. Applications taking advantage of this new global organization are mushrooming. Collaborative work, from its humble beginnings as electronic mail and network newsgroups, is moving into workflow, multimedia, and true distributed environments [25, 12, 6, 5]. Heterogeneous and physically-separated information sources are being linked together. Tasks are being delegated across the network by means of agents [26]. Electronic commerce is possible through secure protocols.

Yet, despite this explosive development, distributed computing itself remains a major challenge. Why is this? A distributed system is a set of autonomous processes, linked together by a network [48, 30, 8]. To emphasize that these processes are not necessarily on the same machine, we call them *sites*. Such a system is fundamentally different from a single process. The system is inherently concurrent and nondeterministic. There is no global information nor global time. Communication delays between processes are unpredictable. There is a large probability of localized faults. The system is shared, so users must be protected from other users and their computational agents.

[*] `seif@sics.se`, Swedish Institute of Computer Science, S-164 28 Kista, Sweden
[†] `pvr@info.ucl.ac.be`, Dép. INGI, Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium
[‡] `perbrand@sics.se`, Swedish Institute of Computer Science, S-164 28 Kista, Sweden
[§] `schulte@dfki.de`, German Research Center for Artificial Intelligence (DFKI), D-66123 Saarbrücken, Germany

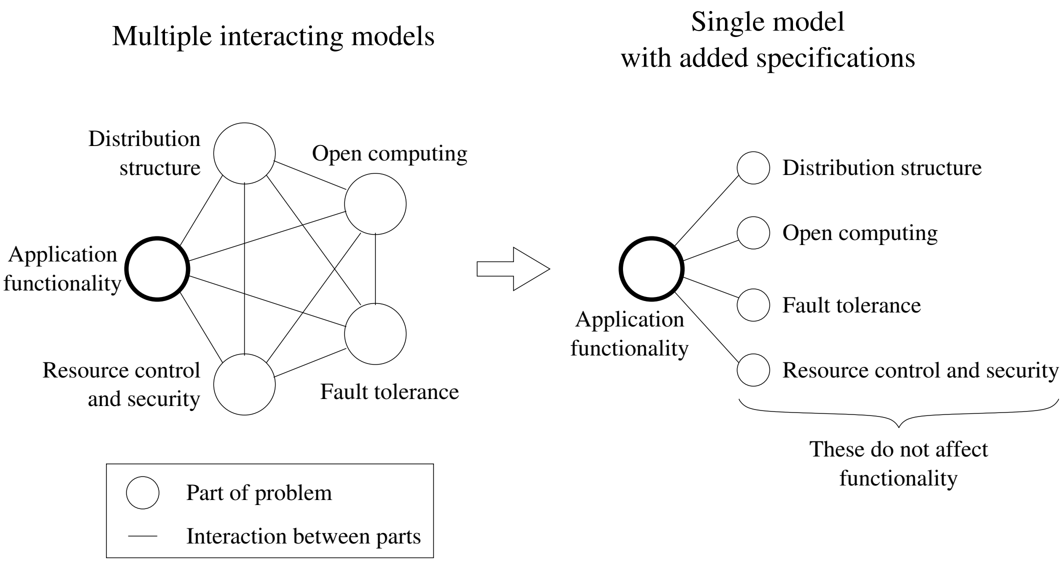


Figure 1: The challenge: simplifying distributed programming

1.1 Identifying the issues

A distributed application should have good perceived behavior, despite the vicissitudes of the underlying system. The application should have good performance, be dependable, and be easily interfaceable with other applications. How can we achieve this?

In the current state of the art, developing a distributed application with these properties requires specialist knowledge beyond that needed to develop an application on a single machine. For example, a new client-server application can be written with Java RMI [33, 34]. An existing application can be connected with another through a CORBA implementation (e.g., Orbix) [37]. Yet in both cases the tools are unsatisfactory. Just to reorganize the distribution structure requires rewriting the application. Because the Java specification does not require time-sliced threads [15], doing such a reorganization in Java may require profound changes to the application. Furthermore, with each new problem that is addressed, e.g., adding a degree of fault tolerance, the complexity of the application increases. To master each, the developer must learn a complex new tool in addition to the environment he or she already knows. A developer experienced only in centralized systems is not prepared.

Some progress has been made in integrating solutions to different problem areas into a single platform. For example, the Ericsson Open Telecom Platform (OTP) [11], based on the Erlang language [4, 54], integrates solutions for both distribution structure and fault tolerance. Erlang is network transparent at the process level, i.e., messages between processes (a form of active objects) are sent in the same way independently of whether the processes are on the same or different sites. The OTP goes far beyond popular platforms such as Java [33, 34] and is being successfully used in commercial telephony products, where reliability is paramount.

The success of the Erlang approach suggests applying it to the other problem areas of distributed computing. We identify four areas, namely distribution structure, open computing, fault tolerance, and security. If the application functionality is included, this means that the application designer has five concerns:

- **Functionality**: what the application does if all effects of distribution are disregarded.
- **Distribution structure**: the partitioning of the application over a set of sites.

- **Open computing**: the ability for independently-written applications to interact with each other in interesting ways.
- **Fault tolerance**: the ability for the application to continue providing its service despite partial failures.
- **Security**: the ability for the application to continue providing its service despite intentional interference. An important part of fault tolerance and security is **resource control**.

A possible approach is to separate the functionality from the other four concerns (see Figure 1). That is, we would like the bulk of an application's code to implement its functionality. Models of the four other concerns should be small and orthogonal additions. Can this approach work? This is a hard question and we do not yet have a complete answer. But some things can be said.

The first step is to separate the functionality from the distribution structure. We say that the system should be both network-transparent and network-aware. A system is *network-transparent* if computations behave in the same way independent of the distribution structure. Applications can be almost entirely programmed without considering the network. A system is *network-aware* if the programmer maintains full control over localization of computations and network communication patterns. The programmer decides where a computation is performed and controls the mobility and replication of data and code. This allows to obtain high performance.

1.2 Towards a solution

We have designed and implemented a language that successfully implements the first step, i.e., it completely separates the functionality from the distribution structure. The resulting language, Distributed Oz, is a conservative extension to the existing centralized Oz language [10]. Porting existing Oz programs to Distributed Oz requires essentially no effort. Why is Oz a good foundation for distributed programming? Because of three properties [46]:

- Oz has a solid formal foundation that does not sacrifice expressiveness or efficient implementation. Oz is based on a higher-order, state-aware, concurrent constraint computation model. Oz appears to the programmer as a concurrent object-oriented language that is every bit as advanced as modern languages such as Java (see Section 3). The current emulator-based implementation is as good or better than Java emulators [20, 19]. Standard techniques for concurrent object-oriented design apply to Oz [28]. Furthermore, Oz introduces powerful new techniques that are not supported by Java [16].
- Oz is a state-aware and dataflow language. This helps give the programmer control over network communication patterns in a natural manner (see Section 4). State-awareness means the language distinguishes between stateless data (e.g., procedures or values), which can safely be copied to many sites, and stateful data (e.g., objects), which at any instant must reside on just one site [52]. Dataflow synchronization allows to decouple calculating a value from sending it across the network [17]. This is important for latency tolerance.
- Oz provides language security. That is, references to all language entities are created and passed explicitly. An application cannot forge references nor access references that have not been explicitly given to it. The underlying representation of language entities is inaccessible to the programmer. Oz has an abstract store with lexical scoping and first-class procedures (see Section 7). These are essential properties to implement a capability-based security policy [49, 53].

Allowing a successful separation of functionality from distribution structure puts severe restrictions on a language. It would be almost impossible in C++ because the semantics are informal and unnecessarily complex and because the programmer has full access to all underlying representations [47]. It is possible in Oz because of the above three properties. So far, it has not been necessary to change the language semantics more than slightly to accommodate distribution.[1] This may change in the future. Furthermore, work is in progress to separate the functionality from the other three concerns. Currently, Distributed Oz provides the language semantics of Oz and complements it in four ways:

- It has constructs to express the distribution structure independently of the functionality (see Section 4). The shared graphic editor of Section 2 is designed according to this approach.
- It has primitives for open computing, based on the concept of *tickets* (see Section 5). This allows independently-running applications to connect and seamlessly exchange data and code.
- It has primitives for orthogonal failure detection and handling, based on the concepts of *handlers* and *watchers* (see Section 6). This allows to build a first level of fault tolerance.
- It supports a capability-based security policy and has primitives for resource control based on the concept of *virtual site* (see Section 7).

In Distributed Oz, developing an application is separated into two independent parts. First, only the logical architecture of the task is considered. The application is written in Oz without explicitly partitioning the computation among sites. One can check the *safety* and *liveness* properties[2] of the application by running it on one site. Second, the application is made *efficient* by specifying the network behavior of its entities. In particular, the mobility of stateful entities (objects) must be specified. For example, some objects may be placed on certain sites, and other objects may be given a particular mobile behavior (such as state caching).

The Distributed Oz implementation extends the Oz implementation with four non-trivial distributed algorithms. Three are designed for specific language entities, namely logic variables, object-records, and object-state. Logic variables are bound with a *variable binding* protocol (see Section 4.2). Object-records are duplicated among sites with a *lazy replication* protocol (see Section 4.3). Object-state moves between sites with a *mobile state* protocol (see Section 4.4). The fourth protocol is a distributed garbage collection algorithm using a credit mechanism (see Section 4.5). Garbage collection is part of the management of shared entities, and it therefore underlies the other three protocols.

1.3 Outline of the article

The rest of this article consists of six parts. Section 2 gives the design of a shared graphic editor in Distributed Oz. It shows how the separation between functionality and distribution works in practice. Section 3 gives an overview of the Oz language and its execution model. Oz has deep roots in the logic programming and concurrent logic programming communities. It is illuminating to show these connections. Section 4 presents Distributed Oz and its architecture, and explains how it separates functionality from distribution structure. The four protocols are highlighted, namely distributed logic variables, lazy replication of object-records, mobility of object-state, and distributed garbage collection. Finally, Sections 5, 6, and 7 discuss open computing, failure detection and handling, and resource control and security. These three sections are more speculative than the others since they describe parts of the system that are still under development.

2 Shared graphic editor

Writing an efficient distributed application can be much simplified by separating the functionality and distribution structure. We have substantiated this claim by designing and implementing a prototype shared graphic editor, an application which is useful in a collaborative work environment. The editor is seen by an arbitrary number of users. We wish the editor to behave like a shared virtual environment. This implies the following set of requirements (see Figure 2). We require that all users

[1] For example, ports have been changed to model asynchronous communication between sites [52].

[2] A fortiori, correctness and termination for nonreactive applications.

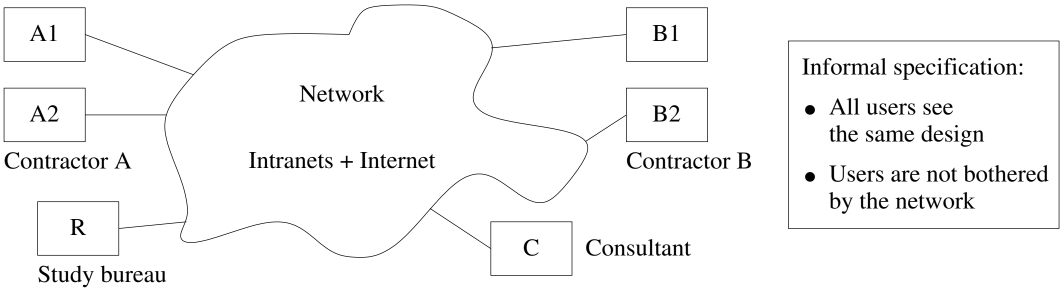


Figure 2: A shared graphic editor

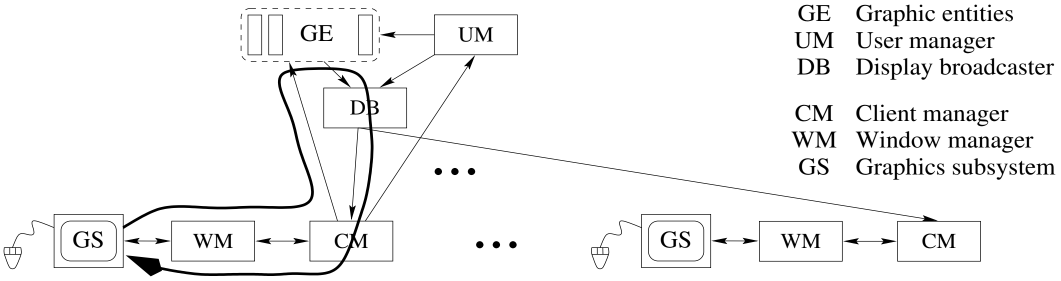


Figure 3: Logical architecture of the graphic editor

be able to make updates to the drawing at any time, that each user sees his or her own updates without any noticeable delays, and that updates must be visible to all users in real time. Furthermore, we require that the same graphic entity can be updated by multiple users. This is useful in a collaborative CAD environment when editing complex graphic designs. Finally, we require that all updates are sequentially consistent, i.e., each user has exactly the same view of the drawing. The last two requirements are what makes the application interesting. Using IP multicast to update each user's visual representation, as is done for example in the LBL Whiteboard application, [3] does not satisfy the last two requirements.

2.1 Logical architecture

Figure 3 gives the logical architecture of our prototype. No assumptions are made about the distribution structure. The drawing state is represented as a set of objects. These objects denote graphic entities such as geometric shapes and freehand drawing pads. When a user updates the drawing, either a new object is created or a message is sent to modify the state of an existing object. The object then posts the update to a display broadcaster. The broadcaster sends the update to all users so they can update their displays. The execution path from user input to display update is shown by the heavy curved line. The users see a shared stream, which guarantees sequential consistency.

New users can connect themselves to the editor at any time using the open computing ability of Distributed Oz. The mechanism is based on "tickets", which are simply text strings (see Section 5). Any Oz process that knows the ticket can obtain a reference to the language entity. The graphic editor creates a ticket for the User Manager object, which is responsible for adding new users. A new user is added by using the ticket to get a reference to the User Manager. The two computations

[3] Available at `http://mice.ed.ac.uk/mice/archive`.

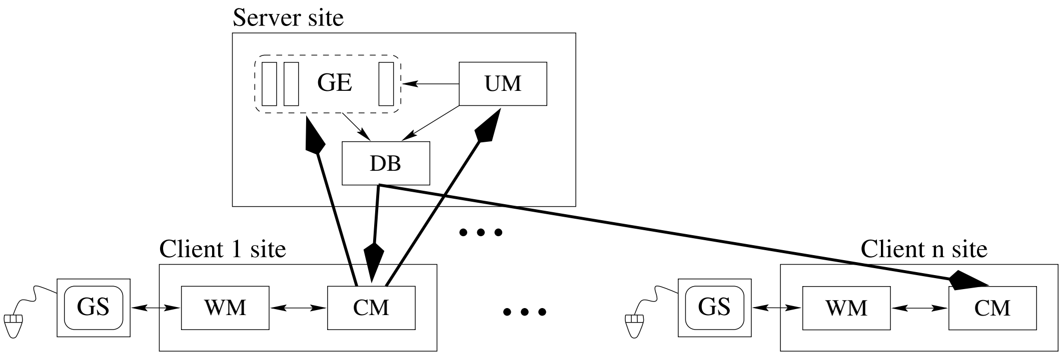


Figure 4: Editor with client-server structure

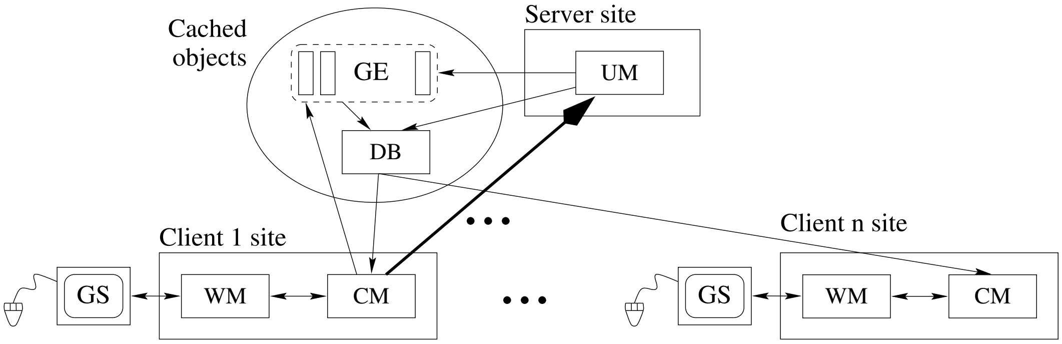


Figure 5: Editor with cached graphic state

then reference the same object. This transparently opens a connection between two sites in the two computations. From that point onward, the computation space is shared. When there are no more references between two sites in a computation, then the connection between them is closed by the garbage collector. Computations can therefore connect and disconnect seamlessly.

2.2 Client-server structure

To realize the design, we have to specify its distribution structure. Figure 4 shows one possibility: a client-server structure. All objects are stationary. They are partitioned among a server site and one site per user. This satisfies all requirements except performance. It works well on low-latency networks such as LANs, but performance is poor when users who are far apart, e.g., in Sweden and Japan, try to draw freehand sketches or any other graphic entity that needs continuous feedback. This is because a freehand sketch consists of many small line segments being drawn in a short time. In our implementation, up to 30 motion events per second are sent from the graphics subsystem to the Oz process. Each line segment requires updating the drawing pad state and sending this update to all users. If the state is remote, then the latency for one update is often several hundred milliseconds or more, with a large variance.

2.3 Cached graphic state

To solve the latency problem, we change the distribution structure (see Figure 5). We refine the design to represent the graphic state and the display broadcaster as freely mobile ("cached") objects rather than stationary objects. The effect of this refinement is that parts of the graphic state are cached at sites that modify them. Implementing the refinement requires changing some of the calls that create new objects. In all, less than 10 lines of code out of 500 have to be changed. With these changes, freehand sketches do not need any network operations to update the local display, so performance is satisfactory. Remote users see the sketch being made in real time, with a delay equal to the network latency. How is this magic accomplished? It is simple: whenever an object is invoked on a site, then the mobile state protocol first makes the object's state pointer local to the site (see Section 4.4). The object invocation is therefore a local operation.

2.4 Push objects and transaction objects

More refined editor designs can take advantage of additional distribution behaviors of objects. For example, the design with cached objects suffers from two problems:

- Users who simultaneously modify different graphic entities will interfere with each other through the display broadcaster. The latter will bounce between user sites, causing delays in updating the displays. This problem can be solved by using a *push object*, which multicasts state updates to all sites that reference the object. One possibility is to make the display broadcaster into a push object, thus maintaining sequential consistency while taking advantage of a multicast network protocol. Another possibility is to make each graphic entity into a push object. In this case, the users may see inconsistent drawings.
- If a user wishes to modify a graphic entity, there is an initial delay while the graphic entity's state is cached on the user site. This problem can be solved by using a *transaction object*, which does the state update locally, while requesting a global lock on the object. The state update will eventually be confirmed or rejected.

Both push and transaction objects maintain consistency of object updates: the object is defined by a sequence of states. It follows that there is still one graphic state and updates to it are sequentially consistent. The editor therefore still supports collaborative design. What changes is how the state sequence is seen and how it is created.

Updating the editor to use either or both of these object types may require changing its specification or logical architecture. For example, the specification may have to be relaxed slightly, temporarily allowing incorrect views. This illustrates the limits of network-transparent programming. It is not possible in general to indefinitely improve the performance of a given specification and logical architecture by changing the distribution structure. At some point, one or both of the specification and architecture must be changed.

2.5 Final comments

Designing the shared graphic editor illustrates the two-part approach for building applications in Distributed Oz. First, build and test the application using stationary objects. Second, reduce latency by carefully selecting a few objects and changing their mobility behavior. Because of transparency, this can be done with quite minor changes to the code of the application itself. This can give good results in many cases. To obtain the very best performance, however, it may be necessary to change the application's specification or architecture.

In both the stationary and mobile designs, fault tolerance is a separate issue that must be taken into account explicitly. It can be done by recording on a reliable site a log of all display events. Crashed users disappear, and new users are sent a compressed version of the log. Primitives for fault tolerance are given in Section 6.

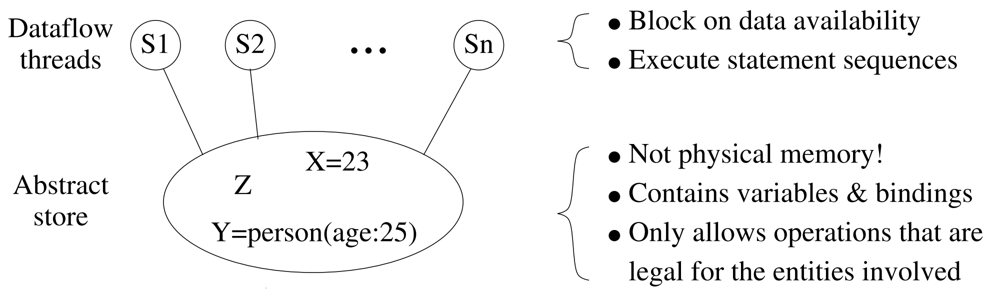


Figure 6: Computation model of OPM

```
S ::=  S S                                              Sequence
    |  X=f(l1:Y1 ... ln:Yn) |                           Value
       X=<number> | X=<atom> | {NewName X}
    |  local X1 ... Xn in S end | X=Y                   Variable
    |  proc {X Y1 ... Yn} S end | {X Y1 ... Yn}         Procedure
    |  {NewCell Y X} | {Exchange X Y Z} | {Access X Y}  State
    |  case X==Y then S else S end                      Conditional
    |  thread S end | {GetThreadId X}                   Thread
    |  try S catch X then S end | raise X end           Exception
```

Figure 7: Kernel language of OPM

In general, mobile objects are useful both for fine-grain mobility (caching of object state) as well as coarse-grain mobility (explicit transfer of groups of objects). The key ability that the system must provide is transparent control of mobility, i.e., control that is independent of the object's functionality. Sections 3.2 and 4 explain briefly how this is done in Distributed Oz. A full explanation is given in [52].

3 Oz

Oz is a rich language built from a small set of powerful ideas. This section attempts to situate Oz among its peers. We summarize its programming model and we compare it with Prolog and with concurrent logic languages.

The roots of Oz are in concurrent and constraint logic programming. The goal of the Oz project is to provide a firm foundation for *all* facets of computation, not just for a declarative subset. The semantics should be fully defined and bring the operational aspects out into the open. For example, concurrency and stateful execution make it easy to write programs that interact with the external world [19]. True higher-orderness results in compact, modular programs [1]. First-class computation spaces allow to program inference engines within the system. For example, it is easy to program multiple concurrent first-class Prolog top levels, each with its own search strategy [41].

Section 3.1 summarizes the Oz programming model, including the kernel languages and the abstractions built on top of it. Section 3.2 illustrates Oz by means of a nontrivial example, namely the implementation of remote method invocation. Section 3.3 compares Oz and Prolog. Finally, Section 3.4 gives the history of Oz from a concurrent logic programming viewpoint.

3.1 The Oz programming model

The basic computation model is an abstract store observed by dataflow threads (see Figure 6). A thread executes a sequence of statements and blocks on the availability of data. The store is not physical memory. It only allows operations that are legal for the entities involved, i.e., no type casting or address calculation. The store has three compartments: the constraint store, containing variables and their bindings, the procedure store, containing procedure definitions, and the cell store, containing mutable pointers ("cells"). The constraint and procedure stores are monotonic, i.e., information can only be added to them, not changed or removed. Threads block on availability of data in the constraint store.

The threads execute a kernel language called Oz Programming Model (OPM) [44]. We briefly describe the OPM constructs as given in Figure 7. Statement sequences are reduced sequentially inside a thread. Values (records, numbers, etc.) are introduced explicitly and can be equated to variables. All variables are logic variables, declared in an explicit scope defined by the **`local`** construct. Procedures are defined at run-time with the **`proc`** construct and referred to by a variable. Procedure applications block until their first argument refers to a procedure. State is created explicitly by `NewCell`, which creates a *cell*, an updatable pointer into the constraint store. Cells are updated by `Exchange` and read by `Access`. Conditionals use the keyword **`case`** and block until the condition is true or false in the constraint store.[4] Threads are created explicitly with the **`thread`** construct and have their own identifier. Exception handling is dynamically scoped and uses the **`try`** and **`raise`** constructs.

Full Oz is defined by transforming all its statements into this basic model. Full Oz supports idioms such as objects, classes, reentrant locks, and ports [44, 52]. The system implements them efficiently while respecting their definitions. We define the essence of these idioms as follows. For clarity, we have made small conceptual simplifications. Full definitions may be found in [16].

- **Object**. An object is essentially a one-argument procedure `{Obj M}` that references a cell, which is hidden by lexical scoping. The cell holds the object's state. The argument `M` indexes into the method table. A method is a procedure that is given the message and the object state, and calculates the new state.

- **Class**. A class is essentially a record that contains the method table and attribute names. When a class is defined, multiple inheritance conflicts are resolved to build its method table. Unlike Java, classes in Oz are pure values, i.e., they are stateless.

- **Reentrant lock**. A reentrant lock is a one-argument procedure `{Lck P}` used for explicit mutual exclusion, e.g., of method bodies in objects used concurrently. `P` is a zero-argument procedure defining the critical section. Reentrant means that the same thread is allowed to reenter the lock. Calls to the lock may therefore be nested. The lock is released automatically if the thread in the body terminates or raises an exception that escapes the lock body.

- **Port**. A port is an asynchronous channel that supports many-to-one communication. A port `P` encapsulates a stream `S`. A stream is a list with unbound tail. The operation `{Send P M}` adds `M` to the end of `S`. Successive sends from the same thread appear in the order they were sent.

3.2 Oz by example

It is not the purpose of this article to give a complete exposition of Oz. Instead, we present Oz by means of a nontrivial example program that is interesting in its own right. We show how to implement active objects in Oz, and as a corollary, we show that the same program implements remote method invocation in Distributed Oz. An active object is an object with an associated thread (or process), much like an actor or concurrent logic process. Invoking a method in an active object is

[4] The keyword **`if`** is reserved for constraint applications.

```
proc {NewStationary Class Init ?StatObj}
   Obj={New Class Init}
   S P={NewPort S}
   N={NewName}
in
   thread
     {ForAll S
      proc {$ M#R}
         thread
            try {Obj M} R=N
            catch E then R=E end
         end
      end}
   end
   proc {StatObj M}
      R in
      {Send P M#R}
      case R==N then skip
      else raise R end
      end
   end
end
```

Figure 8: RMI part 1: Create a stationary object from any class

```
class Counter
   attr i
   meth init i<-0 end
   meth inc i<- @i+1 end
   meth get(X) X=@i end
   meth error raise some_error end end
end

Obj={NewStationary Counter init}
{Obj inc}
{Obj inc}
{Print {Obj get($)}}
try {Obj error} catch X then {Print X} end
```

Figure 9: RMI part 2: A stationary counter object

	SICStus Prolog	Oz
Constraints	Incremental solver with tell	Incremental solver with ask, tell
Control	Backtracking and coroutining	Explicit dataflow threads, encapsulated search
Higher-order	Call, assert	First-class procedures with lexical scoping
State	Objects, mutables, assert	Objects, cells

Table 1: Oz and Prolog

done by explicitly sending a message to the associated thread. As we will see, this kind of object has a well-defined distribution behavior in Distributed Oz. Because threads are stationary in Distributed Oz, the objects also are stationary and reside on their creation site. Invoking the object from a remote site behaves exactly like a remote method invocation.

In Distributed Oz, objects are mobile by default and will execute on the invoking site, inside the invoking thread. This is implemented by a lightweight mobility protocol that serializes the path of the object's state pointer among the invoking sites (see Section 4). One way to make an object stationary, is to wrap the object inside a port and create a thread that invokes the object with messages read from the port's stream. The object is accessed only from this thread, so the object is stationary.

Figure 8 defines the procedure `NewStationary` that implements stationary objects by wrapping them inside a port. It takes a class `Class` and initialization message `Init`, and returns a procedure `StatObj`. The "?" is a comment that denotes an output argument. Inside `NewStationary`, an object `Obj` is created, as well as a port `P` and its associated stream `S`. A thread is created that serves each message appearing on `S`. This is done using the higher-order procedure `{ForAll S Proc}` where `Proc` is a one-argument procedure. The thread waits until a message `M#R` appears on the stream `S` and then executes the procedure call `{Proc M#R}`. The procedure starts a thread that invokes the object with `{Obj M}` and binds `R` either to a unique name `N` denoting normal execution or to an exception `E`. The use of the lexically-scoped new name `N` avoids conflicts with existing exceptions. Let us now consider the procedure `StatObj`. A thread executing `StatObj` sends on the port `P` the pair `M#R` where `M` is the message and `R` is a logic variable for the answer. It suspends on `R` until the corresponding method is executed successfully or an exception is returned. In the latter case the exception is reraised in the thread executing `StatObj`.

We see that Oz allows the programmer to provide generic abstractions that can be used later without concern for their implementation. It is not necessary to understand `NewStationary` in order to use it. This is because the objects it creates have the same Oz semantics as objects created by the standard procedure `New`.

Figure 9 defines a `Counter` class, creates a stationary instance, `Obj`, and sends several messages to `Obj`. Whether `Obj` is created by `NewStationary` or `New`, its language behavior is the same. The `Counter` class does not have any ancestors, therefore no inheritance declaration appears. Each instance of `Counter` has one attribute `i` and four methods. An attribute is a mutable entity that can be accessed and modified from within a method. A method is defined by a method head, which is a record, and a method body, which is a statement. Dynamic binding is supported through the use of **`self`** inside a method body. Accessing the value of an attribute is done by the operator "`@`". Assigning a new value to an attribute is done by the operator "`<-`". Therefore, the method `init` initializes `i` to `0`, the method `inc` increments `i`, the method `get` gets the current value of `i`, and the method `error` raises the somewhat unusual exception `some_error`. Oz has syntactic support for embedding statements in expressions. A statement can be used as an expression by using a "`$`" to mark the result. Therefore `{Print {Obj get($)}}` is equivalent to **`local`** `X` **`in`** `{Obj get(X)} {Print X}` **`end`**.

	Concurrent logic programming	Oz
Constraints	None, except in AKL	Incremental solver with ask, tell
Control	Fine-grain concurrency	Explicit dataflow threads, encapsulated search
Higher-order	Restricted	First-class procedures with lexical scoping
State	Stream-based objects	Objects, cells

Table 2: Oz and concurrent logic programming

3.3 Oz and Prolog

There is a strong sense in which Oz is a successor to Prolog (see Table 1). The Oz system can be used for many of the tasks for which Prolog and constraint logic programming are used today [32, 41, 21, 45]. Like Prolog, Oz has a declarative subset. Like Prolog, Oz has been generalized to arbitrary constraint systems (currently implemented are finite domains and open feature structures). Oz is fully defined and has an efficient implementation competitive with the best emulated Prolog systems [19, 35, 50]. Even though Oz has much in common with Prolog, it is not a Prolog superset. Oz does not have Prolog's reflective syntax (i.e., data and programs have the same syntax), nor does it have the meta-programming facilities (like `call/1`, `assert/1`) or the user-definable syntax (operator declarations).

The foundation of Prolog's success is the high abstraction level of its declarative subset, namely first-order Horn clause logic with SLDNF resolution [29]. What's missing from Prolog is that little attempt is made to give the same foundation to anything *outside* the declarative subset. Two decades of research have resulted in a solid understanding of the declarative subset and only a partial understanding of the rest.[5] This results in two main flaws of Prolog. First, the operational aspects are too deeply intertwined with the declarative. The control is naive (depth-first search) and eager. The interactive top level has a special status: it is lazy, but unfortunately inaccessible to programs. It is lazy because new solutions are calculated upon user request. It is inaccessible to programs, i.e., a program cannot internally set up a query and request solutions lazily. To provide a top level within a program requires programming a meta-interpreter, thus losing an order of magnitude in efficiency. Second, to express anything beyond the declarative subset requires ad hoc primitives that are limited and do not always do the right thing. The `freeze/2` provides coroutining as a limited form of concurrency. The `call/1` and `setof/3` provide only a limited form of higher-orderness. All these problems are solved in Oz.

3.4 Oz and concurrent logic programming

Oz is the latest in a long line of concurrent logic languages. Table 2 compares Oz with concurrent logic programming languages. First experiments with concurrency were done in the venerable IC-Prolog system where coroutining was used to simulate concurrent processes. This led to the Parlog language and Concurrent Prolog. The advent of GHC simplified concurrent logic programming considerably by introducing the notion of *quiet guards*. A clause matching a goal will fire only if the guard is entailed by the constraint store. This formulation and its theoretical underpinning were pioneered by the work of Maher and Saraswat as they gave a solid foundation to concurrent logic programming [31, 40]. On the practical side, the flat versions of Concurrent Prolog and GHC, called FCP and FGHC respectively, were the focus of much work [13, 43]. The KL1 language, derived from FGHC, was implemented in the high-performance KLIC system. This system runs on sequential, parallel, and distributed machines [14]. A number of implementation techniques in the current Distributed Oz system have been borrowed from KLIC, notably the distributed garbage collection algorithm.

[5] The non-declarative aspect has received some attention, e.g., [36, 39, 3].

Kind of entity	Protocol		Entity
Stateless	Replication	Eager	record, procedure, class
		Lazy	object-record
Single assignment	Binding	Eager	logic variable
		Lazy	logic variable
Stateful	Localization	Mobile	cell, object-state
		Stationary	port, thread

Table 3: Semantics of Distributed Oz

An important subsequent development was AKL (Andorra Kernel Language) [23], which added explicit state in the form of ports and provided the first synthesis of concurrent and constraint logic programming. AKL encapsulates search by using nested computation spaces. A computation space is a constraint store with its associated goals. Search is done by allowing procedures to be defined by a sequence of don't-know guarded clauses. These definitions denote disjunctions. When local propagation cannot choose between different disjuncts, then the program is free to try them by cloning the computation space. The initial Oz system, Oz 1, was largely derived from AKL, but added the notions of higher-order procedures, more controllable search by making computation spaces first class, compositional syntax, and the cell primitive for mutable state. Concurrency in Oz 1 is fine-grained. When a statement suspends, a new thread is created that contains only the suspended statement. The main thread is not suspended but continues with the next statement.

All concurrent logic languages up to and including Oz 1 were designed for fine-grained concurrency and implicit exploitation of parallelism. The current Oz language, Oz 2, abandons this model in favor of explicit control over concurrency by means of a thread creation construct. Thread suspension and resumption is still based on dataflow using logic variables. Our experience shows that explicit concurrency makes it easier for the user to control application resources. It allows the language to have an efficient and expressive object-oriented model without sequential state threading within method definitions. It also allows easy incorporation of a conventional exception handling construct into the language, and last but not least a simple debugging model. In the current Oz system concurrency is used mostly to model logical concurrency in the application rather than to increase potential parallelism.

4 Distributed Oz

Distributed Oz has the same language semantics as Oz. Distributed Oz separates application functionality from distribution structure by defining a distributed semantics for all language entities [52, 51, 17, 18]. The distributed semantics extends the language semantics to take into account the notion of site. It defines the network operations invoked when a computation is partitioned on multiple sites. We classify the language entities into three basic types (see Table 3):

- Stateless entities are replicated eagerly (records, procedures, classes) or lazily (object-record).
- Single assignment entities (logic variables) are bound eagerly or lazily [17].
- Stateful entities are localized and are either mobile by default (cell, object-state) or stationary by default (port, thread) [52]. What moves is not the state, but the site that has the right to create the next state. We say that this site has the *state pointer*.[6]

For each of these entities, network operations[7] are predictable, which gives the programmer the ability to manage network communications. In the rest of this section, we present the four distributed algorithms used to implement the language entities. Section 4.1 introduces the concept of *access*

[6] In [52] it is called the *content-edge*.

[7] In terms of the number of network hops.

structure, which models a language entity that is accessible from more than one site. The distributed behavior of a language entity is defined as a protocol between the nodes of its access structure, i.e., as a distributed algorithm. Sections 4.2 explains the uses of distributed logic variables and shows how to bind them with a *variable binding* protocol. Sections 4.3 and 4.4 show how to build mobile objects that have predictable network behavior by using a *lazy replication* protocol for the object-record (explained in Section 4.3) and a *mobile state* protocol for the object-state (explained in Section 4.4). The network behavior of logic variables and objects highlights most clearly the design philosophy of Distributed Oz. Finally, Section 4.5 explains the distributed garbage collection algorithm, which underlies the management of access structures.

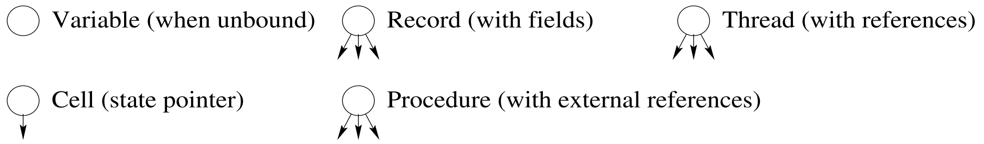


Figure 10: Language entities as nodes in a graph

4.1 The distribution graph

We model distributed execution in a simple but precise manner using the concept of *distribution graph*. 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 10). Objects are compound entities and are explained in Section 4.3.

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 11). 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. An access structure consists of one *proxy node* `Pi` for each site that referenced the original node and one *manager node* `M` for the whole structure. The resulting graph, containing both local nodes and access structures where necessary, is called the *distribution graph*. Most of the example protocol executions in this article use this notation.

Each access structure is given a global address that is unique system-wide. The global address encodes various pieces of information including the manager site. Proxy nodes are uniquely identified by pairs (global address,site). On each site, the global address indexes into a table that refers to the proxy. This allows to enforce the invariant that each site has at most one proxy. Messages are sent between nodes in access structures. In terms of sites, a message is sent from the source node's site to the destination node's site. In the message body, all references are to nodes on the destination site. These nodes are identified by the global addresses of their access structures. When the message arrives, the nodes are looked up in the site table.

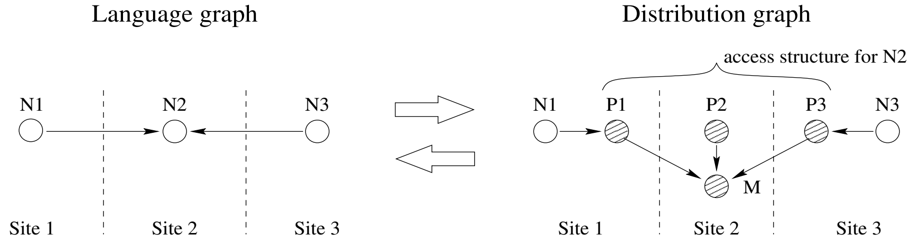
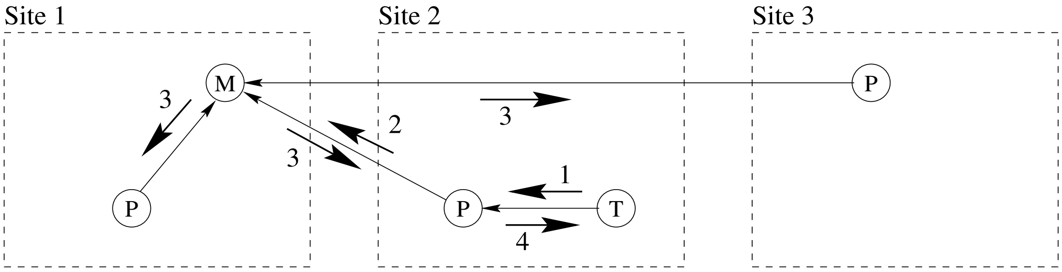


Figure 11: Access structure in the distribution graph



1 Thread initiates binding and blocks
2 Proxy requests binding
3 Manager grants binding & multicasts to all proxies
4 Proxy informs thread, allowing thread to continue

Figure 12: Binding a logic variable

Procedures and other values (records and numbers, etc.) are copied eagerly, i.e., they never result in an access structure.[8] A procedure is only sent once to any site[9] and has only one copy on the site. A procedure consists of a closure and a code block, each of which is given a global address. Messages contain only the global addresses, and upon arrival the missing code blocks and closures are requested immediately.

4.2 Distributed logic variables

Logic variables express dependencies between computations without imposing an execution order. This property can be exploited in distributed computing:

- Two basic problems in distributed computing are latency tolerance and third-party independence. Using logic variables instead of explicit message passing can improve these two aspects of an application with little programming effort.

- Using logic variables, common distributed programming idioms can be expressed in a network-transparent manner that results in optimal or near-optimal message traffic.

These benefits are realized due to a practical distributed algorithm for rational tree unification, which is used to bind logic variables [17]. The algorithm is efficiently implemented in the Distributed Oz system as two parts: a local algorithm and a distributed algorithm. Most of the work of unification is done locally. The distributed algorithm does only variable binding. We briefly describe it here.

The two basic operations on logic variables are binding and waiting until bound. A logic variable `X` can be bound to a data structure or to another variable. The algorithm is the same in both cases. If many bindings to `X` are initiated concurrently (from one or more sites), then only one will succeed. The other bindings are then retried with the entity to which `X` is bound. By default, binding is *eager*. That is, the new value is immediately sent to all sites that know about `X`. This means that a bound variable is guaranteed to eventually disappear from the system.

We illustrate the binding algorithm with an example. In the distribution graph, a logic variable shows up as an access structure. Figure 12 shows a variable that exists on three sites. A thread on site 2 initiates a binding of the variable by informing its proxy (message 1) and then blocking. The proxy asks the manager to bind the variable (message 2). The manager informs all proxies of

[8] In [52] there is a variant design in which objects are procedures and all procedures are copied lazily.
[9] Unless a garbage collection removes it.

the binding (message 3), thus binding the variable eagerly. When a proxy receives the binding, it informs all waiting threads (message 4). The threads then continue execution.

Logic variables can have different distributed behaviors, as long as network-transparency is satisfied in each case. A logic variable is *eager* by default. This gives maximal latency tolerance and third-party independence. However, this may cause the binding to be sent to sites that do not need it. We say that a logic variable is *lazy* if its value is only sent to a site when the site requests it (e.g., when a thread needs the value). A lazy variable has better message complexity, i.e., fewer messages are used. In some cases, e.g., implementing barrier synchronization using a short-circuit technique, lazy variables are preferable. Eager and lazy variables obey the same distributed unification algorithm, differing only in the scheduling of one reduction rule [17]. Distributed Oz currently only implements eager variables; with a minor change it can do both. A programmer annotation can then decide whether a variable is eager or lazy.

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

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

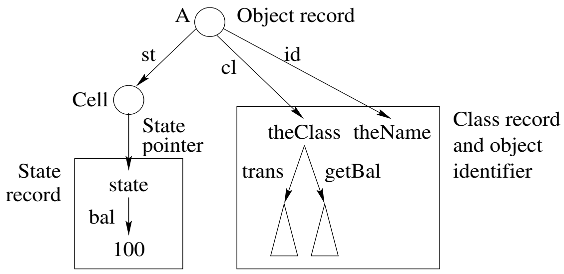


Figure 13: An object with one attribute and two methods

4.3 Mobile objects

Objects in Distributed Oz obey a lightweight object migration protocol that preserves centralized object semantics and allows for precise prediction of network behavior. Existing systems with mobile objects do not use such an algorithm. They move the objects by creating a chain of forwarding references [34, 24, 7]. This chain is short-circuited when a message is sent or after a given time delay. This gives good average-case number of network hops when moving an object, but very bad worst-case number of hops. A design principle of Distributed Oz is for third-party dependencies to always eventually disappear. Using chains is therefore unacceptable, since the intermediate links in the chain may last indefinitely. Instead, we have designed the mobility protocol presented here, which has a much-improved worst-case behavior.

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 cell (containing the 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 13 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 cell, whose state pointer refers to the object's state record. 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 cell (i.e., updating the state pointer), the object state can be updated.

Figure 14 shows an object `A` that is local to Site 1. There are no references to `A` from any other sites. Figure 15 shows an object `A` with one remote reference. The 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 a message referencing `A` arrives on Site 2, then a proxy node `Pa2` is created there.

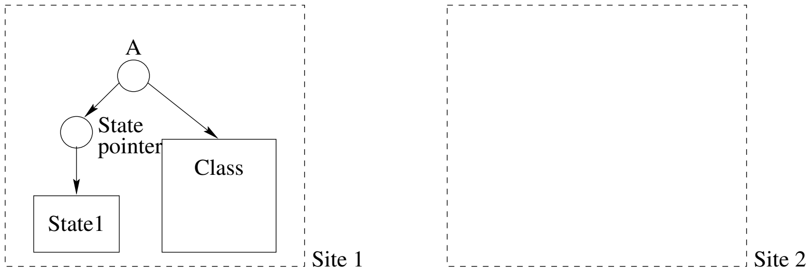


Figure 14: A local object

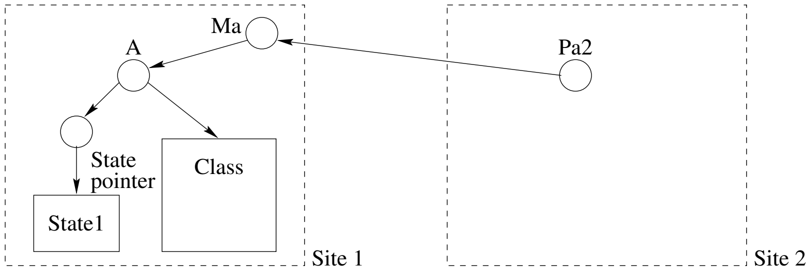


Figure 15: A global object with one remote reference

Figure 16 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 cell, with a manager `Mc` and one proxy `Pc1`. The class record is copied eagerly and does not have a unique global address. A message containing the class record and a cell proxy is sent to Site 2. The object's state remains on Site 1.

Figure 17 shows what happens when the message arrives. A second proxy `Pc2` is created for the cell. The class record is copied to Site 2 and proxy `Pa2` becomes the object-record `A`. The site table now refers to the object-record. The mobile state protocol (see Section 4.4) then atomically transfers the cell to Site 2. Because of the site table, any further messages to Site 2 containing references to the object will immediately refer to the local copy of the object-record, without requiring any additional network operations.

Figure 18 shows what happens after the cell is transferred to Site 2. The new state, `State2`, is created on Site 2 and will contain the updated object state after the method finishes. The old state, `State1`, may continue to exist on Site 1 but the cell no longer points to it.

Figure 19 shows what happens if Site 1 invokes the object again. The cell 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 cell no longer points to it.

There are several interesting things going on here. First, the object is always executed locally. The cell's 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 cell's manager node. This simplifies the protocol but introduces a dependency on the manager site. A more complicated protocol (not shown here) can remove this dependency [52].

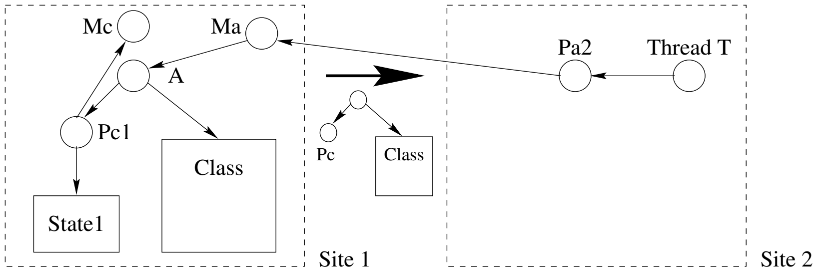


Figure 16: The object is invoked remotely (1)

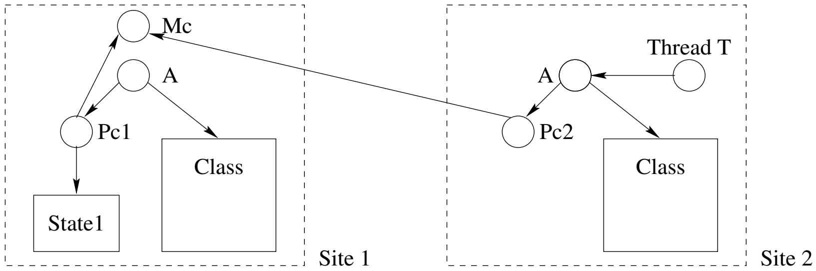


Figure 17: The object is invoked remotely (2)

4.4 Mobile state

The freely mobile objects shown in Section 4.3 are composite 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. At all times, the state pointer of the object's cell access structure exists at exactly one proxy, or is in transit between two proxies. The protocol that moves the state pointer, the *mobile state* protocol, is particularly interesting. This protocol must guarantee consistency between consecutive states. If the consecutive states are on different sites, this requires an atomic transfer of the state pointer between the sites. A site that wants the state pointer requests it from the cell manager, and the latter sends a forwarding command to the site that has the state pointer. Therefore the manager needs to store only one piece of information, namely the site containing the state pointer [52].

Figure 20 shows a cell `C` referenced from two sites. The cell's state pointer is on Site 1 and Site 2 requests it when thread `T` does the operation {`Exchange C X Y`}. It suffices to know that the exchange is an atomic swap that sets the new state to `Y` (i.e., to `State2`) and initiates a binding of `X` to the old state `State1`.

Figure 21 shows (a) proxy `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 has (or 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.

Figure 22 shows `Pc1` sending to `Pc2` a `Content` message containing the old state, `State1`. The old state may still exist on Site 1 but `Pc1` no longer has a pointer to it. Figure 23 shows the final situation. `Pc2` has the state pointer, which points to `State2`. `X` is bound to `State1`.

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, cell exchanges (updates of the state pointer) are done in a globally consistent order.

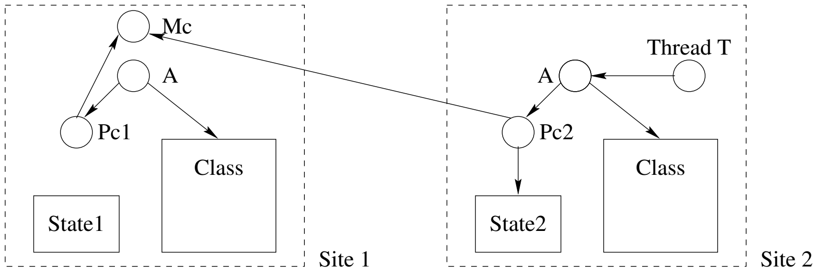


Figure 18: The object is invoked remotely (3)

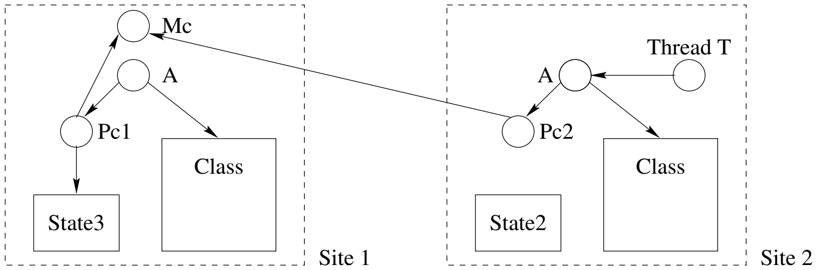


Figure 19: The object moves back to Site 1

4.5 Distributed garbage collection

Access structures are built and managed automatically when language entities become remotely referenced. This happens whenever messages exchanged between nodes on different sites contain references to other nodes. If the reference is to a local node, then the memory management layer converts the local node into an access structure. We say the local node is *globalized*. While the message is in the network, the access structure consists of a manager and one proxy. When the message arrives at the destination site, then a new proxy is created there. Access structures can reduce in size and disappear completely through garbage collection.

Distributed garbage collection is implemented by two cooperating mechanisms: a local garbage collector per site and a distributed credit mechanism to reclaim global addresses. A local garbage collector informs the credit mechanism when a node is no longer referenced on its site. Conversely, the credit mechanism informs the local garbage collector when a node is no longer remotely referenced. Local collectors can be invoked at any time independently of other sites. The roots of local garbage collection are all nodes on its site that are reachable from non-suspended thread nodes or are remotely referenced.

A global address is reclaimed when the node that it refers to is no longer remotely referenced. This is done by the credit mechanism, which is informed by the local garbage collectors. This scheme recovers all garbage except for cross-site cycles. The only cross-site cycles in our system occur between different objects or cells. Since records and procedures are both replicated, cycles between them will be localized to single sites. The credit mechanism does not suffer from the memory or network inefficiencies of previous reference-counting schemes [38].

We summarize briefly the basic ideas of the credit mechanism. Each global address is created with an integer (its *debt*) representing the number of *credits* that have been given out to other sites and to messages. Any site or message that contains the global address must have at least one credit for the global address. The creation site is called the *owner*. All other sites are called *borrowers*. A node is remotely referenced if and only if its debt is nonzero.

Initially there are no borrowers, so the owner's debt is zero. The owner lends credits to any site or message that refers to the node and increments its debt each time by the number of credits lent.

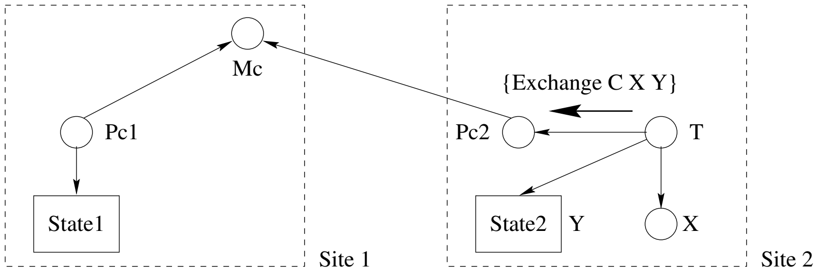


Figure 20: A cell referenced from two sites

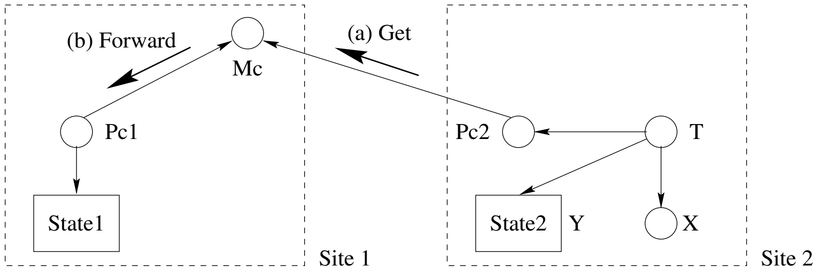


Figure 21: (a) Site 2 requests the state pointer; (b) Site 1 is asked to forward it

When a message arrives at a borrower, its credits are added to the credits already present. When a message arrives at the owner, its credits are deducted from the owner's debt. When a borrower no longer locally references a node then all its credits are sent back to the owner. This is done by the local garbage collector. When the owner's debt is zero then the node is only locally referenced, so its global address will be reclaimed.

Consider the case of a cell access structure. The manager site is the owner, and all other sites with cell proxies are borrowers. A proxy disappears when no longer locally referenced. It then sends its credit back to the manager. If the proxy contains the state pointer, then the state pointer is transferred back to the manager site as well. Remark that this removes a cross-site cycle within the cell access structure. When the manager recovers all its credit then it disappears, and the cell becomes a local cell again. When the local cell has no local references, then it is reclaimed. If the local cell becomes global again (because a message referring to it is sent across the network), then a new manager is created, completely unrelated to the reclaimed one.

5 Open computing

We say a distributed system is *open* if independently-running applications can interact in interesting ways [9]. In general, this means that the system must have common ground, in the form of common frameworks or languages, that applications can use to interact. Typical examples are common information formats for exchanging information, common protocols for electronic commerce, etc. As a first requirement, applications must be able to establish connections with computations that have been started independently across the net. A second requirement is that applications should be able to initiate new distributed computations.

5.1 Connections and tickets

Oz uses a ticket-based mechanism to establish connections between independent Oz sites. In the complete system, both the tickets and the connections require a mechanism for providing security

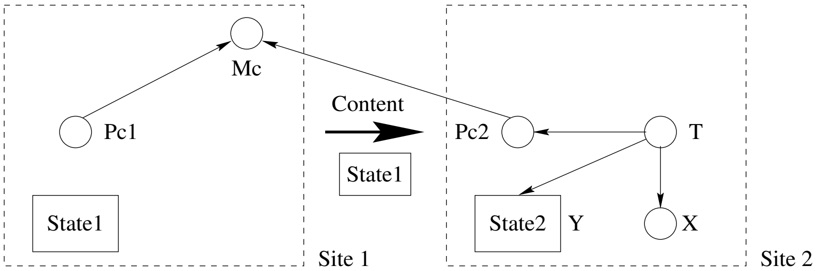


Figure 22: Site 1 has sent the state pointer to Site 2

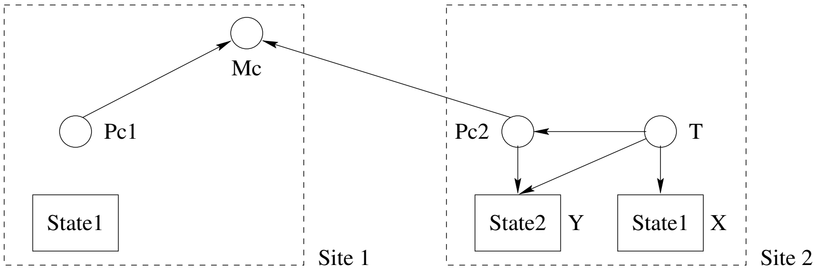


Figure 23: Site 2 has the state pointer

(see Section 7). In this section, we explain the basic mechanism without discussing security issues. One site (called the server site) creates a ticket with which other sites (called client sites) can establish a connection. The ticket is a character string which can be stored and transported through all media that can handle text, e.g., phone lines, electronic mail, paper, and so forth.

The ticket identifies both the server site and the language entity to which a remote reference will be made. Independent connections can be made to different entities on the same site. Establishing a connection has two effects. First, the sites connect by means of a network protocol (e.g., TCP). Second, in the Oz computation space, a reference is created on the client site to a language entity on the server site. The second effect can be implemented by various means, i.e., by passing a zero-argument procedure, by unifying two variables, or by passing a port which is then used to send further values. Once an initial connection is established, then further connections as desired by applications can be built from the programming abstractions available in Oz. For example, it is possible to define a class `C` on one site, pass `C` to another site, define a class `D` inheriting from `C` on that site, and pass `D` back to the original site. This works because Distributed Oz is fully transparent.

Oz features two different types of tickets: one-shot tickets that are valid for establishing a single connection only (one-to-one connections), and many-shot tickets that allow multiple connections to the ticket's server (many-to-one connections). One-to-one connections are useful when connecting to newly-started compute servers (see Section 5.2). Many-to-one connections are useful in collaborative applications such as the shared graphic editor of Section 2. Multiple users need to connect to the same application in order to contribute to a common design.

We sketch a small example for one-to-one connections:

Server	**Client**
`STkt={Connection.offer X}`	`CTkt={QueryUser}`
`{PutOnWebPage STkt}`	`X={Connection.take CTkt}`
`{ProcessData X}`	`X=data(···)`

The server offers `X` with the system procedure `Connection.offer`, which is part of the module `Connection`. This procedure takes the offered value and returns a new one-shot ticket `STkt`, which is made available on a Web page. The user reads the page, retrieves the ticket, and types it in at the client site, which puts it in variable `CTkt`. The system procedure `Connection.take` then returns a reference to `X`, which becomes a shared reference between the client and server. In this example, `X` is a shared logic variable. It could have been any language entity, e.g., an object or a port. The client binds `X` and the server reads its value. This passes information from the client to the server.

5.2 Remote compute servers

Distributed applications mainly fall into two categories. In the first category, applications involve geographically-distributed resources. For example, in the shared graphic editor of Section 2 the users are the distributed resources. In the second category, applications use multiple networked computers to increase computation speed. These applications are often structured as a single master computation that coordinates a set of slave computations. The actual computation is carried out by the slaves.

To support the second category, Oz provides the ability to create remote compute servers, which are accessible as Oz objects. This is implemented using the ticket mechanism. After the compute server has been set up, it can be given tasks to do in the form of procedures. The following example shows one way to use a compute server:

```
S={New RemoteServer init('wallaby.ps.uni-sb.de')}
{Print {S run(fun {$} 4+5 end $)}}
```

The remote server site is started on the computer with Internet address `'wallaby.ps.uni-sb.de'`. The `run` method takes a zero-argument function that is executed at the remote server. In the example, two numbers are added and the result `9` is returned to the original site, where it is printed.

Setting up a remote server is done in two steps. Assume that a site wishes to create a remote server and then become a client to the server. First, the potential client creates an independently-running Oz site with the help of the operating system.[10] Second, a connection is established between the potential client and the remote server. This is done by passing a one-shot ticket from the potential client to the server. The server takes a logic variable that has been offered by the client and binds it to a stationary procedure (see Section 3.2). This stationary procedure is used on the client side to implement the `run` method shown in the example above.

As an application of this idea, we are currently investigating distributed search engines for solving combinatorial constraint problems. First experiments show encouraging speedup. Oz supports the two aspects of distributed search engines in a powerful way: search engines can be easily programmed in Oz [42], and the language supports distributed computing well.

6 Failure detection and handling

An application is *fault-tolerant* if it can continue to fulfill its specification despite accidental failures of its components. How can one write such applications in Distributed Oz? The theory of fault-tolerant systems explains how to construct such systems as layers of abstractions [22]. Very little work has been done to integrate these abstractions into a language platform so that (1) a fault-tolerant system can be built within the platform, and (2) the integration is orthogonal to the language entities. Most of the language work has been concentrated in the areas of persistence and transactions, by adding models of these concepts to an existing language. It is possible, however, to support fault tolerance in a much simpler way.

We extend the system to support partial failure of sites and individual language entities, and to detect and handle failure of language entities. We provide the means for the programmer to decide what action to take upon failure. This is done by installing “handlers” or “watchers” on individual language entities (see below). These are invoked when a failure occurs. No irrevocable decision is taken by the system; the handlers and watchers are free to take any course of action. In this way,

[10] In the current system, a remote site is started by the Unix remote shell command (`rsh`).

we intend to build a first fault-tolerant layer using the redundancy that comes from having multiple sites in the system. This gives fault tolerance even in the absence of persistence. More refined fault tolerance based on persistence and transactions will be added later.

6.1 The containment principle

Fault tolerance is a property that crosses abstraction boundaries [27]. An example will make this clear. Most existing systems (we include applications) do not handle *time* correctly. What they do is let a lower layer make an irrevocable decision, in the form of a *time-out* that does not let the system continue. Say there is a time-out in a lower layer, for example in the transport layer (TCP) of the network interface. This time-out crosses all abstraction boundaries to appear at the top level, i.e., to the user. Usually, a window is opened asking confirmation to abort the application. The user does not have the possibility to communicate back to the timed-out layer. This greatly limits the flexibility of the system. It should be possible to build a system where the user can decide to wait, avoiding an abort, or to abort immediately without waiting. In most cases, neither of these possibilities is offered. Sometimes one possibility is offered, thus improving the perceived quality of the system. A hard-mounted resource in the NFS file system offers the first possibility. The Stop button in a Web browser offers the second possibility.

This leads to a principle of containment: "abnormal" behavior of any layer should be containable by a higher layer. Therefore the abnormal layer should not make any irrevocable decisions, such as aborting execution, unless this is desired by the programmer. The programmer decides which higher layer is competent to handle the problem. The higher layer should be able to take any reasonable course of action. In our example, this means that time-out should not be a wired-in property of a system. The programmer should decide whether or not to have a time-out, and what to do after a given time has elapsed (one of the possibilities being to continue waiting).

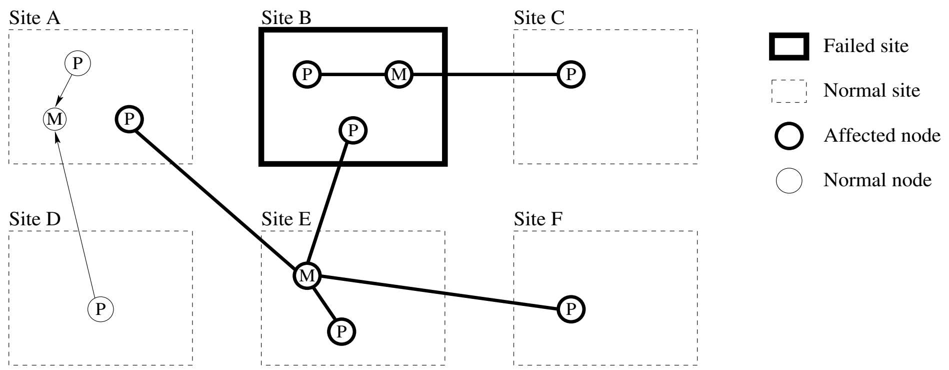


Figure 24: Remote detection of site failure in Distributed Oz

6.2 Failures in the distribution graph

The external cause of a failure in Distributed Oz is the failure of one or more sites or of part of the network. This shows up in the distribution graph at the level of access structures. We say an access structure is *affected* if it has at least one node on a failed site or if it has at least one link across a failed network. An affected access structure can in many cases continue to work normally, e.g., an object can still be used even if it has a remote reference on a failed site. An access structure is *failed* if normal sites can no longer do operations on it. This happens if a crucial part of the access structure, e.g., the manager node, is inaccessible because it is on a failed site or across a failed network. Figure 24 shows a system that covers six sites. Site B has failed; sites A, C, E, and F have affected nodes; and site D has only normal nodes. We assume that sites are designed to satisfy the fail-stop property, i.e., site failures happen instantly and are permanent. Networks may have temporary failures, i.e., the network may return to normal. An access structure can therefore have both temporary and permanent failures.

6.3 Handlers and watchers

Distributed Oz detects failure at the level of access structures, which shows up in the language as single language entities, i.e., objects, variables, and ports. The default behavior is that an attempted operation on an entity blocks indefinitely if there is a problem in doing the operation. Any other behavior must be specified explicitly by the programmer. We propose to do this by installing handlers and watchers on the entity. A *handler* is invoked if an error is detected when trying to do an operation on the entity (lazy detection). A *watcher* is invoked when an error is detected for an entity, even if no operation is attempted on the entity (eager detection).

The semantics of handlers and watchers is simple. If an operation is attempted on a failed entity, then the operation is replaced by a call of the handler, if one exists with a valid trigger condition. If the system discovers that an entity has failed, then every watcher with a matching trigger condition is run in a newly-created thread. Handlers and watchers have two arguments, namely the failed entity itself and information about the type of error.

Handlers may be installed on entities per site and per thread. Per site and per entity, there is at most one site-based handler and at most one thread-based handler per thread. Thread-based handlers override site-based handlers, i.e., where both apply only the thread-based handler is invoked. Watchers may be installed on entities per site. There may be any number of watchers per entity on a given site.

Handlers and watchers are installed by builtins with the following three arguments: the entity, control information, and the handler or watcher itself, which is a two-argument procedure. The control information gives the type of error for which the handler or watcher should be invoked. In the case of handlers, the control information also stipulates whether the handler is installed on a site or thread basis, and whether after handler invocation the operation should be retried.

6.4 Classifying possible failures

Failure detection distinguishes between four classes of failure. A failure can be either temporary or permanent. These can be further subdivided into home and foreign failures. Home failures prevent the current site from performing operations on the entity, while foreign failures indicate that there is a problem among other sites sharing the entity preventing them from performing operations on the entity. Handlers are triggered on home failures. Watchers may be triggered on home as well as foreign failures. Foreign failures give the site an indication that the there is more than network latency behind a lack of activity by other sites.

6.5 Distributed garbage collection with failures

If a site fails, then credit is lost for all affected access structures whose manager is still working. These access structures will not be reclaimed unless we introduce another idea. A technique that is successfully being used in existing systems, e.g., Java RMI [34], is a *lease-based* reference-counting mechanism. This technique can also be used together with the credit mechanism. Any site that has credit for an access structure must periodically renew its lease by sending a message to the manager. If the manager does not receive at least one renewal message within a given time period, then the manager can be reclaimed.

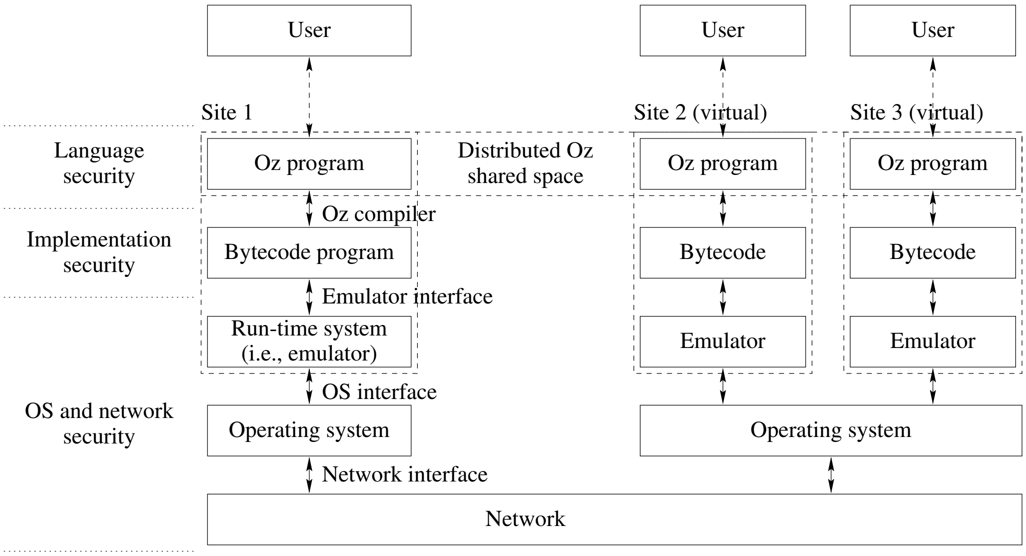


Figure 25: Security issues in Distributed Oz

7 Resource control and security

An application is *secure* if it can continue to fulfill its specification despite intentional (i.e., malicious) failures of its components. Resource control and security are global issues, i.e., they cross abstraction boundaries [2], just like fault tolerance. The issues must therefore be addressed at each layer. We briefly discuss what can be done in Distributed Oz. Fault tolerance and security have much in common [27], including the reliance on containment and redundancy. But they focus on very different classes of failures. For example, a crucial part of security is resource control because exhausting resources is a common technique to provoke intentional failures ("denial of service" attacks). Although important, resource control is less critical for fault tolerance.

Resources are conveniently divided into site and network resources. Site resources include computational resources (memory/processor) and other resources such as file systems and peripherals. The same site resources normally appear in some form at each site layer, i.e., Oz program, emulator, and operating system. In a similar way, security issues appear at each layer (see Figure 25):

- **Language security** is a property of the language. It guarantees that computations and data are protected from adversaries that stay within the language.
- **Implementation security** is a property of the language implementation in the process. It protects computations and data from adversaries who attempt to interfere with compiled programs, i.e., with the Oz bytecode.
- **Operating system and network security** are properties of the operating system and network. They protect computations and data from adversaries who attempt to interfere with the internals of the Oz emulator and run-time system within an operating system process, and who attempt to interfere with the operating system and the network. Network security is available through secure TCP/IP.

7.1 Language security

We provide language security by giving the programmer the means to restrict access to data. Data are represented as references to entities in an abstract shared computation space. The space is *ab-*

```
% Create new module ROpen that looks like the standard Open
% but allows only reading and only in the given directory Dir:
proc {NewReadInDir Dir ROpen}
   class ROpenF
      attr fd
      meth init(name:FN)
         % Should verify absence of '..' and '/' in FN!
         fd <- {New Open.file init(name:Dir#FN)}
      end
      meth read(list:CL)
         {@fd read(list:CL)}
      end
   end
in
   ROpen=open(file:ROpenF)
end

% Give limited rights to an untrusted object:
SandboxOpen = {NewReadInDir "/usr/home/untrusted_foreign/"}
{UntrustedObj setopenmodule(SandboxOpen)}
```

Figure 26: Capabilities in Oz

stract because it provides a well-defined set of basic operations. In particular, unrestricted access to memory is forbidden.[11] One can only access data to which one has been given an explicit reference.

A reference to a procedure or an object behaves as a capability. Because of lexical scoping and first-class procedures [1], it is possible to create new capabilities that encapsulate existing ones, thus possibly limiting their rights. For example, Figure 26 shows how to give an object limited rights to a file system. Calling `{NewReadInDir Dir ROpen}` creates a new module `ROpen`, which behaves exactly like the system module `Open`, except that it only allows to read files and only in the given directory `Dir`.

Capabilities do not solve all problems in security [53]. They have inherent weaknesses. First, the authorization to do something is given very early, namely when the capability is given and not when the operation is attempted. Second, a capability can be forwarded to anyone and it will continue working. Therefore, a capability-based mechanism needs to be extended–for example with access control based on the identity of the capability's current possessor.

7.2 Implementation security

Two important issues in implementation security are site integrity and resource control. These issues appear when code is directly or indirectly passed between sites. For example, sending a procedure to a compute server to execute it (direct) or invoking a method of a mobile object (indirect). The foreign code is not necessarily trustworthy. The importing site should be protected from being corrupted by malicious foreign code, i.e., by invalid Oz bytecode. This is very difficult in general. Typical techniques are bytecode verification and authenticating compilers [49].

The foreign code should be limited in its ability to use the site's computational resources. Monopolizing the processor may starve the site's other concurrent activities. Excessive memory use may exhaust the site's memory, which being extremely difficult to recover from, would effectively crash the site.

[11] For example, both examining data representations (type casts) and calculating addresses (pointer arithmetic) are forbidden.

The foreign code should be given controlled access to other site resources. Obviously, untrusted code cannot be given unlimited access to site-specific resources such as file systems. It is possible, but not practical, to forbid access to all site-specific resources (just as it is not practical to forbid all access to basic computation resources!). Better is to provide limited capabilities.

7.3 Virtual sites

To partially provide implementation security in Distributed Oz, we propose the mechanism of *virtual sites* (see Figure 25). A site can spawn slave virtual sites, which behave exactly like standard sites except that the master monitors and controls the slaves. If the slave crashes then the master is notified but not otherwise affected. The master controls slaves' resources, including their computational resources and other resources such as access to file systems. For example, the slave site might be given the possibility to create and delete files in one specific directory but nowhere else.

Within the limitations imposed by the master, a virtual site behaves almost exactly the same as an ordinary site. It may share Oz entities with the master site or any other site. The difference is that the virtual site shares the same machine. Communication is more efficient since there is no network layer. To take advantage of the protection and resource control mechanisms of the operating system, a slave site will normally live in a different process than its master.

Virtual sites can be used to exploit the computational resources of shared-memory multiprocessors. Simply allocate one processor per virtual site. Because communication overheads are lower, this is more efficient than parallelism over the network. Whether the parallelism leads to an effective speedup depends of course on the communication overhead.

8 Conclusion

Distributed programming is of major importance today, yet it remains difficult. We present a design for a distributed programming language, Distributed Oz, that fully separates an application's functionality from its distribution structure. Distributed Oz is a conservative extension to the existing centralized Oz language. Oz is a concurrent object-oriented language that is state-aware and that has dataflow synchronization. Oz programs can be ported almost immediately to Distributed Oz, which is implemented and publicly available. We are experimenting with distributed applications including collaborative tools, compute servers, and techniques for using centralized applications in distributed settings [6].

Distributed Oz is very much work in progress. We present preliminary designs that conservatively extend the language with models for open computing, fault tolerance, and resource control. These designs are being implemented and extended.

Acknowledgements

We thank the numerous people at SICS, DFKI, and UCL that have contributed to Distributed Oz. We thank Donatien Grolaux for suggesting transaction objects. Seif Haridi and Per Brand are supported by the Swedish national board for industrial and technical development (NUTEK) and SICS. Christian Schulte is supported by the Bundesminister für Bildung, Wissenschaft, Forschung und Technologie (FKZ ITW 9601) and the Esprit Working Group CCL-II (EP 22457).

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass, 1985.

[2] Edward G. Amoroso. *Fundamentals of Computer Security Technology*. Prentice-Hall, 1994.

[3] James Andrews. The logical semantics of the Prolog cut. In *International Logic Programming Symposium (ILPS 95)*, December 1995.

[4] J. Armstrong, M. Williams, C. Wikström, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, N.J., 1996.

[5] Tomas Axling, Seif Haridi, and Lennart Fahlen. Concurrent constraint programming virtual reality applications. In the *2nd International Conference on Military Applications of Synthetic Environments and Virtual Reality* (*MASEVR 95*), Stockholm, Sweden, 1995. Defence Material Administration.

[6] Per Brand, Nils Franzen, Erik Klintskog, and Seif Haridi. A platform for constructing virtual spaces. In *Virtual Worlds and Simulation Conference (VWSIM '98)*, January 1998.

[7] Luca Cardelli. A language with distributed scope. *ACM Transactions on Computer Systems*, 8(1):27–59, January 1995. Also appeared in POPL 95.

[8] Randy Chow and Theodore Johnson. *Distributed Operating Systems and Algorithms*. Addison-Wesley, San Francisco, Calif., 1997.

[9] Jon Crowcroft. *Open Distributed Systems*. University College London Press, London, U.K., 1996.

[10] DFKI Oz version 2.0, 1998. Available at `http://www.ps.uni-sb.de`.

[11] Ericsson. *Open Telecom Platform—User's Guide, Reference Manual, Installation Guide, OS Specific Parts*. Telefonaktiebolaget LM Ericsson, Stockholm, Sweden, 1996.

[12] François Fluckiger. *Understanding Networked Multimedia: Applications and Technology*. Prentice-Hall, 1995.

[13] Institute for New Generation Computer Technology, editor. *Fifth Generation Computer Systems 1992*, volume 1,2. Ohmsha Ltd. and IOS Press, 1992. ISBN 4-274-07724-1.

[14] Tetsuro Fujise, Takashi Chikayama, Kazuaki Rokusawa, and Akihiko Nakase. KLIC: A portable implementation of KL1. In *Fifth Generation Computing Systems (FGCS '94)*, pages 66–79, December 1994.

[15] James Gosling and Henry McGilton. The Java language environment. White paper, Sun Microsystems, Mountain View, Calif., May 1996.

[16] Seif Haridi. Tutorial of Oz 2. Available at `http://www.sics.se/~seif/oz.html`, 1996.

[17] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Using logic variables in distributed computing. Submitted to ACM TOPLAS, February 1998.

[18] Seif Haridi, Peter Van Roy, and Gert Smolka. An overview of the design of Distributed Oz. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*, pages 176–187, Maui, Hawaii, USA, July 1997. ACM Press.

[19] Martin Henz. *Objects for Concurrent Constraint Programming*, volume 426 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, November 1997.

[20] Martin Henz. *Objects in Oz*. PhD thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, June 1997.

[21] Joxan Jaffar and Michael Maher. Constraint logic programming: A survey. *J. Log. Prog.*, 19/20:503–581, May/July 1994.

[22] Pankaj Jalote. *Fault Tolerance in Distributed Systems*. PTR Prentice-Hall, 1994.

[23] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In *International Symposium on Logic Programming*, pages 167–183, October 1991.

[24] 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.

[25] Setrag Khoshafian and Marek Buckiewicz. *Introduction to Groupware, Workflow, and Workgroup Computing*. J. Wiley and Sons, 1995.

[26] Michael Knapik and Jay Johnson. *Developing Intelligent Agents for Distributed Systems*. McGraw-Hill, 1998.

[27] J. C. Laprie. Dependability: A unifying concept for reliable computing and fault tolerance. In *7th International Conference on Distributed Computing Systems*, pages 129–146, September 1987.

[28] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.

[29] John Lloyd. *Foundations of Logic Programming, Second Edition*. Springer-Verlag, 1987.

[30] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, Calif., 1996.

[31] Michael Maher. Logic semantics for a class of committed-choice programs. In *International Conference on Logic Programming (ICLP 87)*, pages 858–876, May 1987.

[32] Martin Müller, Tobias Müller, and Peter Van Roy. Multiparadigm programming in Oz. In *Workshop on the Future of Logic Programming, International Logic Programming Symposium (ILPS 95)*, December 1995.

[33] Sun Microsystems. *The Java Series*. Sun Microsystems, Mountain View, Calif., 1996. Available at `http://www.aw.com/cp/javaseries.html`.

[34] Sun Microsystems. *The Remote Method Invocation Specification*. Sun Microsystems, Mountain View, Calif., 1997. Available at `http://www.javasoft.com`.

[35] Michael Mehl, Ralf Scheidhauer, and Christian Schulte. An Abstract Machine for Oz. In *Programming Languages, Implementations, Logics, and Programming (PLILP '95)*, 1995.

[36] Lee Naish. *Negation and Control in Prolog*. Springer-Verlag, 1986. Lecture Notes in Computer Science, vol. 238.

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

[38] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In the *International Workshop on Memory Management*, Lecture Notes in Computer Science, vol. 986, pages 211–249, Berlin, September 1995. Springer-Verlag.

[39] Andreas Podelski and Gert Smolka. Operational semantics of constraint logic programs with coroutining. In *International Conference on Logic Programming (ICLP 95)*, pages 449–463, 1995.

[40] Vijay Saraswat and Martin Rinard. Concurrent constraint programming. In *POPL*, pages 232–245, January 1990.

[41] Christian Schulte. Oz Explorer: A visual constraint programming tool. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 286–300, Leuven, Belgium, July 1997. The MIT Press.

[42] Christian Schulte. Programming constraint inference engines. In Gert Smolka, editor, *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 519–533, Schloß Hagenberg, Austria, October 1997. Springer-Verlag.

[43] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):413–510, September 1989.

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

[45] Gert Smolka, Christian Schulte, and Jörg Würtz. *Finite Domain Constraint Programming in Oz: A Tutorial*. Programming Systems Lab, German Research Center for Artificial Intelligence (DFKI), January 1998. In Oz 2 system documentation. Available at `http://www.ps.uni-sb.de`.

[46] Gert Smolka, Christian Schulte, and Peter Van Roy. PERDIO—Persistent and distributed programming in Oz. BMBF project proposal. Available at `http://www.ps.uni-sb.de`, February 1995.

[47] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1997.

[48] Gerard Tel. *An Introduction to Distributed Algorithms*. Cambridge University Press, Cambridge, United Kingdom, 1994.

[49] Tommy Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997.

[50] Peter Van Roy. 1983–1993: The wonder years of sequential Prolog implementation. *J. Log. Prog.*, 19/20:385–441, May/July 1994.

[51] Peter Van Roy, Seif Haridi, Per Brand, and Gert Smolka. Three moves are not as bad as a fire. In the *Workshop on Internet Programming Languages, International Conference on Computer Languages (ICCL 98)*, May 1998.

[52] 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.

[53] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *16th Symposium on Operating System Principles*, October 1997.

[54] 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.

Contents