Distributed Programming in Mozart - A Tutorial Introduction

Peter Van Roy
Seif Haridi
Per Brand

Version 1.3.2
June 15, 2006

Abstract

This tutorial shows how to write efficient and robust distributed applications with the Mozart programming system. We first present and motivate the distribution model and the basic primitives needed for building distributed applications. We then progressively introduce examples of distributed applications to illustrate servers, agents, mobility, collaborative tools, fault tolerance, and security.

The tutorial is suitable for Oz programmers who want to be able to quickly start writing distributed programs. The document is deliberately informal and thus complements the other Oz tutorials and the research papers on distribution in Oz.

The Mozart programming system has been developed by researchers from DFKI (the German Research Center for Artificial Intelligence), SICS (the Swedish Institute of Computer Science), the University of the Saarland, UCL (the Université catholique de Louvain), and others.

THE MATERIAL IN THIS DOCUMENT IS INCOMPLETE. THIS DOCUMENT WILL EVENTUALLY BE SUPERSEDED BY A COMPLETE DOCUMENT. IN THE MEANTIME, WE RECOMMEND LOOKING AT THE DISTRIBUTED PROGRAMMING (DP) CATEGORY OF THE MOGUL ARCHIVE AND READING CHAPTER 11 OF THE BOOK 'CONCEPTS, TECHNIQUES, AND MODELS OF COMPUTER PROGRAMMING'.

Credits

Mozart logo by Christian Lindig

License Agreement

Contents

1

Introduction

Fueled by the explosive development of the Internet, distributed programming is becoming more and more popular. The Internet provides the first steps towards a global infrastructure for distributed applications: a global namespace (URLs) and a global communications protocol (TCP/IP). Both platforms based on the Java language and on the CORBA standard take advantage of this infrastructure and have become widely-used. On first glance, one might think that distributed programming has become a solved problem. But this is far from the case. Writing efficient, open, and robust distributed applications remains much harder than writing centralized applications. Making them secure increases the difficulty by another quantum leap. The abstractions offered by Java and CORBA, for example the notion of distributed object, provide only rudimentary help. The programmer must still keep distribution and fault-tolerance strongly in mind.

The Mozart platform is the result of three years of research into distributed programming and ten years of research into concurrent constraint programming. The driving goal is to separate the fundamental aspects of programming a distributed system: application functionality, distribution structure, fault tolerance, security, and open computing.

The current Mozart release completely separates application functionality from distribution structure, and provides primitives for fault-tolerance, open computing, and partial support for security. Current research is focused on completing the separation for fault tolerance and open computing, which will be offered in upcoming releases. Future research will focus on security and other issues.

This tutorial presents many examples of practical programs and techniques of distributed programming and fault-tolerant programming. The tutorial also gives many examples of useful abstractions, such as cached objects, stationary objects, fault-tolerant stationary objects, mobile agents, and fault-tolerant mobile agents, and shows how easy it is to develop new abstractions in the Mozart platform.

Essentially all the distribution abilities of Mozart are given by four modules:

- The module `Connection`[1] provides the basic mechanism (known as *tickets*) for active applications to connect with each other.

[1] Chapter *Connecting Computations:* `Connection`, *(System Modules)*

- The module `Remote`[2] allows an active application to create a new site (local or remote operating system process) and connect with it. The site may be on the same machine or a remote machine.
- The module `Pickle`[3] allows an application to store and retrieve arbitrary stateless data from files and URLs.
- The module `Fault`[4] gives the basic primitives for fault detection and handling.

The first three modules, `Connection`[5], `Remote`[6], and `Pickle`[7], are extremely simple to use. In each case, there are just a few basic operations. For example, `Connection`[8] has just two basic operations: offering a ticket and taking a ticket.

The fourth module, `Fault`[9], is the base on which fault-tolerant abstractions are built. The current module provides complete fault-detection ability for both site and network failures and has hooks that allow to build efficient fault-tolerant abstractions within the Oz language. This release provides a few of the most useful abstractions to get you started. The development of more powerful ones is still ongoing research. They will be provided in upcoming releases.

This tutorial gives an informal but precise specification of both the distribution model and the failure model. The tutorial carefully indicates where the current release is incomplete with respect to the specification (this is called a *limitation*) or has a different behavior (this is called a *modification*). All limitations and modifications are explained where they occur and they are also listed together at the end of the tutorial (see Chapter 5).

We say two or more applications are *connected* if they share a reference to a language entity that allows them to exchange information. For example, let Application 1 and Application 2 reference the same object. Then either application can call the object. All low-level data transfer between the two applications is automatic; from the viewpoint of the system, it's just one big concurrent program where one object is being called from more than one thread. There is never any explicit message passing or encoding of data.

The Mozart platform provides much functionality in addition to distribution. It provides an interactive development environment with incremental compiler, many tools including a browser, debugger, and parser-generator, a C++ interface for developing dynamically-linked libraries, and state-of-the-art constraint and logic programming support. We refer the reader to the other tutorials and the extensive system documentation.

[2] Chapter *Spawning Computations Remotely:* `Remote`, *(System Modules)*
[3] Chapter *Persistent Values:* `Pickle`, *(System Modules)*
[4] Chapter *Detecting and Handling Distribution Problems:* `Fault`, *(System Modules)*
[5] Chapter *Connecting Computations:* `Connection`, *(System Modules)*
[6] Chapter *Spawning Computations Remotely:* `Remote`, *(System Modules)*
[7] Chapter *Persistent Values:* `Pickle`, *(System Modules)*
[8] Chapter *Connecting Computations:* `Connection`, *(System Modules)*
[9] Chapter *Detecting and Handling Distribution Problems:* `Fault`, *(System Modules)*

2

Distribution Model

The basic difference between a distributed and a centralized program is that the former is partitioned among several sites. We define a *site* as the basic unit of geographic distribution. In the current implementation, a site is always one operating system process on one machine. A multitasking system can host several sites. An Oz language entity has the same language semantics whether it is used on only one site or on several sites. We say that Mozart is *network-transparent*. If used on several sites, the language entity is implemented using a distributed protocol. This gives the language entity a particular distributed semantics in terms of network messages.

The *distributed semantics* defines the network communications done by the system when operations are performed on an entity. The distributed semantics of the entities depends on their type. The distribution model gives well-defined distributed semantics to all Oz language entities.

The distributed semantics has been carefully designed to give the programmer full control over network communication patterns where it matters. The distributed semantics does the right thing by default in almost all cases. For example, procedure code is transferred to sites immediately, so that sites never need ask for procedure code. For objects, the developer can specify the desired distributed semantics, e.g., mobile (cached) objects, stationary objects, and stationary single-threaded objects. Section 2.1 defines the distributed semantics for each type of language entity, Section 2.2 explains more about what happens at sites, and Section 2.3 outlines how to build distributed applications.

2.1 Language entities

2.1.1 Objects

The most critical entities in terms of network efficiency are the objects. Objects have a state that has to be updated in a globally-consistent way. The efficiency of this operation depends on the object's distributed semantics. Many distributed semantics are possible, providing a range of trade-offs for the developer. Here are some of the more useful ones:

- *Cached object*: Objects and cells are cached by default–we also call this "mobile objects". Objects are always executed locally, in the thread that invokes the method. This means that a site attempting to execute a method will first fetch the object, which requires up to three network messages. After this, no further messages are needed as long as the object stays on the site. The object will not move as long as execution stays within a method. If many sites use the object, then it will travel among the sites, giving everyone a fair share of the object use.

The site where the object is created is called its *owner site*. A reference to an object on its owner site is called an *owner* or *owner node*. All other sites referencing the object are *proxy sites*. A remote reference to an object is called a *proxy* or a *proxy node*. A site requesting the object first sends a message to the owner site. The owner site then sends a forwarding request to the site currently hosting the object. This hosting site then sends the object's state pointer to the requesting site.

The class of a cached object is copied to each site that calls the object. This is done lazily, i.e., the class is only copied when the object is called for the first time. Once the class is on the site, no further copies are done.

- *Stationary object (server)*: A stationary object remains on the site at which it was created. Each method invocation uses one message to start the method and one message to synchronize with the caller when the method is finished. Exceptions are raised in the caller's thread. Each method executes in a new thread created for it on the object's site. This is reasonable since threads in Mozart are extremely lightweight (millions can be created on one machine).

- *Sequential asynchronous stationary object*: In this object, each method invocation uses one message only and does not wait until the method is finished. All method invocations execute in the same thread, so the object is executed in a completely sequential way. Non-caught exceptions in a method are ignored by the caller.

Deciding between these three behaviors is done when the object is created from its class. A cached object is created with `New`, a stationary object is created with `NewStat`, and an sequential asynchronous stationary object is created with `NewSASO`. A stationary object is a good abstraction to build servers (see Section 3.2.3) and fault-tolerant servers. It is easy to program other distribution semantics in Oz. Chapter 3 gives some examples.

2.1.2 Other stateful entities

The other stateful language entities have the following distributed semantics:

- *Thread*: A thread actively executes a sequence of instructions. The thread is stationary on the site it is created. Threads communicate through shared data and block when the data is unavailable, i.e., when trying to access unbound logic variables. This makes Oz a data-flow language. Threads are *sited* entities (see Section 2.1.5).

- *Port*: A port is an asynchronous many-to-one channel that respects FIFO for messages sent from within the same thread. A port is stationary on the site it is created, which is called its *owner site*. The messages are appended to a stream on the port's site. Messages from the same thread appear in the stream in the same order in which they were sent in the thread. A port's stream is terminated by a *future* (see Section 2.1.3).

Sending to a local port is always asynchronous. Sending to a remote port is asynchronous except if all available memory in the network layer is in use. In that case, the send blocks. The network layer frees memory after sending data across the network. When enough memory is freed, the send is continued. This provides an end-to-end flow control.

Oz ports, which are a language concept, should not be confused with Unix ports, which are an OS concept. Mozart applications do not need to use Unix ports explicitly except to communicate with applications that have a Unix port interface.

- *Cell*: A cell is an updatable pointer to any other entity, i.e., it is analogous to a standard updatable variable in imperative languages such as C and Java. Cells have the same distributed semantics as cached objects. Updating the pointer may need up to three network messages, but once the cell is local, then further updates do not use the network any more.
- *Thread-reentrant lock*: A thread-reentrant lock allows only a single thread to enter a given program region. Locks can be created dynamically and nested recursively. Locks have the same distributed semantics as cached objects and cells. This implements a standard distributed mutual exclusion algorithm.

2.1.3 Single-assignment entities

An important category of language entities are those that can be assigned only to one value:

- *Logic variable*: Logic variables have two operations: they can be bound (i.e., assigned) or read (i.e., wait until bound). A logic variable resembles a single-assignment variable, e.g., a `final` variable in Java. It is more than that because two logic variables can be bound together even before they are assigned, and because a variable can be assigned more than once, if always to the same value. Logic variables are important for three reasons:
  - They have a more efficient protocol than cells. Often, variables are used as placeholders, that is, they will be assigned only once. It would be highly inefficient in a distributed system to create a cell for that case.

    When a logic variable is bound, the value is sent to its *owner site*, namely the site on which it was created. The owner site then multicasts the value to all the proxy sites, namely the sites that have the variable. The current release implements the multicast as a sequence of message sends. That is, if the variable is on *n* sites, then a maximum of *n*+1 messages are needed to bind the variable. When a variable arrives on a site for the first time, it is immediately registered with the owner site. This takes one message.
  - They can be used to improve latency tolerance. A logic variable can be passed in a message or stored in a data structure before it is assigned a value. When the value is there, then it is sent to all sites that need it.
  - They are the basic mechanism for synchronization and communication in concurrent execution. Data-flow execution in Oz is implemented with logic variables. Oz does not need an explicit monitor or signal concept–rather,

logic variables let threads wait until data is available, which is 90% of the needs of concurrency. A further 9% is provided by reentrant locking, which is implemented by logic variables and cells. The remaining 1% are not so simply handled by these two cases and must be programmed explicitly. The reader is advised not to take the above numbers too seriously.

- *Future*: A future is a read-only logic variable, i.e., it can only be *read*, not bound. Attempting to bind a future will block. A future can be created explicitly from a logic variable. Futures are useful to protect logic variables from being bound by unauthorized sites. Futures are also used to distribute constrained variables (see Section 2.1.5).

- *Stream*: A stream is an asynchronous one-to-many communication channel. In fact, a stream is just a list whose last element is a logic variable or a future. If the stream is bound on the owner site, then the binding is sent asynchronously to all sites that have the variable. Bindings from the same thread appear in the stream in the same order that they occur in the thread.

  A port together with a stream efficiently implement an asynchronous many-to-many channel that respects the order of messages sent from the same thread. No order is enforced between messages from different threads.

2.1.4 Stateless entities

Stateless entities never change, i.e., they do not have any internal state whatsoever. Their distributed semantics is very efficient: they are copied across the net in a single message. The different kinds of stateless entities differ in when the copy is done (eager or lazy) and in how many copies of the entity can exist on a site:

- *Records and numbers*: This includes lists and strings, which are just particular kinds of records. Records and numbers are copied eagerly across the network, in the message that references them. The same record and number may occur many times on a site, once per copy (remember that integers in Mozart may have any number of digits). Since these entities are so very basic and primitive, it would be highly inefficient to manage remote references to them and to ensure that they exist only once on a site. Of course, records and lists may refer to any other kind of entity, and the distributed semantics of that entity depends on its type, not on the fact of its being inside a record or a list.

- *Procedures, functions, classes, functors, chunks, atoms, and names*: These entities are copied eagerly across the network, but can only exist once on a given site. For example, an object's class contains the code of all the object's methods. If many objects of a given class exist on a site, then the class only exists there once.

  Each instance of all the above (except atoms) is globally unique. For example, if the same source-code definition of a procedure is run more than once, then it will create a different procedure each time around. This is part of the Oz language semantics; one way to think of it is that a new Oz name is created for every procedure instance. This is true for functions, classes, functors, chunks, and of course for names too. It is not true for atoms; two atoms with the same print name are identical, even if created separately.

- *Object-records*: An object is a composite entity consisting of an object-record that references the object's features, a cell, and an internal class. The distribution semantics of the object's internal class are different from that of a class that is referenced explicitly independent of any object. An object-record and an internal class are both chunks that are copied lazily. I.e., if an object is passed to a site, then when the object is called there, the object-record is requested if it is missing and the class is requested if it is missing. If the internal class already exists on the site, then it is not requested at all. On the other hand, a class that referenced explicitly is passed eagerly, i.e., a message referencing the class will contain the class code, even if the site already has a copy.

In terms of the language semantics, there are only two different stateless language entities: procedures and records. All other entities are derived. Functions are syntactic sugar for procedures. Chunks are a particular kind of record. Classes are chunks that contain object methods, which are themselves procedures. Functors are chunks that contain a function taking modules as arguments and returning a module, where a module is a record.

2.1.5 Sited entities

Entities that can be used only on one site are called *sited*. We call this site their *owner site* or *home site*. References to these entities can be passed to other sites, but they do not work there (an exception will be raised if an operation is attempted). They work only on their owner site. Entities that can be used on any site are called *unsited*. Because of network transparency, unsited entities have the same language semantics independent of where they are used.

In Mozart, all sited entities are modules, except for a few exceptional cases listed below. Not all modules are sited, though. A *module* is a record that groups related operations and that possibly has some internal state. The modules that are available in a Mozart process when it starts up are called *base modules*. The base modules contain all operations on all basic Oz types. There are additional modules, called *system modules*, that are part of the system but loaded only when needed. Furthermore, an application can define more modules by means of functors that are imported from other modules. A *functor* is a module specification that makes explicit the resources needed by the module.

All base modules are unsited. For example, a procedure that does additions can be used on another site, since the addition operation is part of the base module `Number`. Some commonly-used base modules are `Number`, `Int`, and `Float` (operations on numbers), `Record` and `List` (operations on records and lists), and `Procedure`, `Port`, `Cell`, and `Lock` (operations on common entities).

Due to limitations of the current release, threads, dictionaries, arrays, and spaces are sited even though they are in base modules. These entities will become unsited in future releases.

When a reference to a constrained variable (finite domain, finite set, or free record) is passed to another site, then this reference is converted to a *future* (see Section 2.1.3). The future will be bound when the constrained variable becomes determined.

We call *resource* any module that is either a system module or that imports directly or indirectly from a system module. All resources are sited. The reason is that they contain state outside of the Oz language. This state is either part of the emulator or external to the Mozart process. Access to this state is limited to the machine hosting the Mozart process. Some commonly-used system modules are `Tk` and `Browser` (system graphics), `Connection` and `Remote` (site-specific distributed operations), `Application` and `Module` (standalone applications and dynamic linking), `Search` and `FD` (constraint programming), `Open` and `Pickle` (the file system), `OS` and `Property` (the OS and emulator), and so forth.

2.2 Sites

2.2.1 Controlled system shutdown

A site can be stopped in two ways: normally or abnormally. The normal way is a controlled shutdown initiated by `{Application.exit I}`, where `I` is the return status (see the module `Application`). The abnormal way is a site crash triggered by an external problem. The failure model (see Chapter 4) is used to survive site crashes. Here we explain what a controlled shutdown means in the distribution model.

All language entities, except for stateless entities that are copied immediately, have an owner site and proxy sites. The owner site is always the site on which the entity was created. A controlled shutdown has no adverse effect on any distributed entity whose owner is on another site. This is enforced by the distributed protocols. For example, if a cell's state pointer is on the shutting-down site, then the state pointer is moved to the owner site before shutting down. If the owner node is on the shutting-down site, then that entity will no longer work.

2.2.2 Distributed memory management

All memory management in Mozart is automatic; the programmer does not have to worry about when an entity is no longer referenced. Mozart implements an efficient distributed garbage collection algorithm that reclaims all unused entities except those that form a cycle of references that exists on at least two different owner sites. For example, if two sites each own an object that references the other, then they will not be reclaimed. If the objects are both owned by the same site, then they will be reclaimed.

This means that the programmer must be somewhat careful when an application references an entity on another site. For example, let's say a client references a server and vice versa. If the client wishes to disconnect from the server, then it is sufficient that the server forget all references to the client. This will ensure there are no cross-site cycles.

2.3 Bringing it all together

Does the Mozart distribution model give programmers a warm, fuzzy feeling when writing distributed applications? In short, yes it does. The distribution model has been designed in tandem with many application prototypes and numerous Gedankenexperimenten. We are confident that it is basically correct.

Developing an application is separated into two independent parts. First, the application is written without explicitly partitioning the computation among sites. One can in fact check the correctness and termination properties of the application by running it on one site.

Second, the objects are given distributed semantics to satisfy the geographic constraints (placement of resources, dependencies between sites) and the performance constraints (network bandwidth and latency, machine memory and speed). The large-scale structure of an application consists of a graph of threads and objects, which access resources. Threads are created initially and during execution to ensure that each site does the desired part of the execution. Objects exchange messages, which may refer to objects or other entities. Records and procedures, both stateless entities, are the basic data structures of the application–they are passed between sites when needed. Logic variables and locks are used to manage concurrency and data-flow execution. See Section 3.3 for more information on how to organize an application.

Functors and resources are the key players in distributed component-based programming. A functor specifies a software component. A functor is stateless, so it can be transparently copied anywhere across the net and made persistent by pickling on a file (see the module `Pickle`[1]). A functor is linked on a site by evaluating it there with the site resources that it needs (see the modules `Module`[2] and `Remote`[3]). The result is a new resource, which can be used as is or to link more functors. Our goal is for functors to be the core technology driving an open community of developers, who contribute to a growing global pool of useful components.

[1] Chapter *Persistent Values:* `Pickle`, *(System Modules)*

[2] Chapter *Module Managers:* `Module`, *(System Modules)*

[3] Chapter *Spawning Computations Remotely:* `Remote`, *(System Modules)*

Basic Operations and Examples

3.1 Global naming

There are two kinds of global names in Oz:

- Internal references, i.e., that can exist only *within* an Oz computation space. They are globally unique, even for references existing before connecting with another application. All data structures in Oz are addressed through these references; they correspond roughly to pointers and network pointers in mainstream languages, but they are protected from abuse (as in Java). See Section 2.1 for more information on the distribution semantics of these references. In most cases, you can ignore these references since they don't affect the language semantics. In this section we will not talk any more of these references.
- External references, i.e., that can exist *anywhere*, i.e., both inside and outside of an Oz computation space. They are also known as external global names. They are represented as character strings, and can therefore be stored and communicated on many different media, including Web pages, Oz computation spaces, etc. They are needed when a Mozart application wants to interact with the external world.

This section focuses on external global names. Oz recognizes three kinds, namely tickets, URLs, and hostnames:

- A *ticket* is a string that references any language entity inside a running application. Tickets are created within a running Oz application and can be used by active applications to connect together (see module `Connection`[1]).
- A *URL* is a string that references a file across the network. The string follows the standard URL syntax. In Mozart the file can be a *pickle*, in which case it can hold any kind of stateless data–procedures, classes, functors, records, strings, and so forth (see module `Pickle`[2]).
- A *hostname* is a string that refers to a host (another machine) across the network. The string follows the standard DNS syntax. An application can use the hostname to start up a Mozart process on the host (see module `Remote`[3]).

For maximum flexibility, all three kinds can be represented as virtual strings inside Oz.

[1] Chapter *Connecting Computations:* `Connection`, *(System Modules)*

[2] Chapter *Persistent Values:* `Pickle`, *(System Modules)*

[3] Chapter *Spawning Computations Remotely:* `Remote`, *(System Modules)*

3.1.1 Connecting applications by means of tickets

Let's say Application 1 has a stream that it wants others to access. It can do this by creating a ticket that references the stream. Other applications then just need to know the ticket to get access to the stream. Tickets are implemented by the module `Connection`[4], which has the following three operations:

- `{Connection.offer X T}` creates a ticket `T` for `X`, which can be any language entity. The ticket can be taken just once. Attempting to take a ticket more than once will raise an exception.
- `{Connection.offerUnlimited X T}` creates a ticket `T` for `X`, which can be any language entity. The ticket can be taken any number of times.
- `{Connection.take T X}` creates a reference `X` when given a valid ticket in `T`. The `X` refers to exactly the same language entity as the original reference that was offered when the ticket was created. A ticket can be taken at any site. If taken at a different site than where the ticket was offered, then there is network communication between the two sites.

Application 1 first creates a ticket for the stream as follows:

```
declare Stream Tkt in
{Connection.offerUnlimited Stream Tkt}
{Show Tkt}
```

The ticket is returned in `Tkt`. Application 1 then publishes the value of `Tkt` somewhere so that other applications can access it. Our example uses `Show` to display the ticket in the emulator window. We will use copy and paste to communicate the ticket to another application. The ticket looks something like `'x-ozticket://193.10.66.30:9002:SpGK0:U4v/y:s:` Don't worry about exactly what's inside this strange atom. Users don't normally see tickets: they are stored in files or passed across the network, e.g., in mail messages. Application 2 can use the ticket to get a reference to the stream:

```
declare Stream in
{Connection.take
   'x-ozticket://193.10.66.30:9002:SpGK0:U4v/y:s:f:xl'
   Stream}
{Browse Stream}
```

If Application 1 binds the stream by doing `Stream=a|b|c|_` then Application 2's browse window will show the bindings.

3.1.2 Persistent data structures by means of pickles

An application can save any stateless data structure in a file and load it again from a file. The loading may also be done from a URL, used as a file's global name. The module `Pickle` implements the saving and loading and the conversion between Oz data and a byte sequence.

For example, let's define a function and save it:

[4] Chapter *Connecting Computations:* `Connection`, *(System Modules)*

```
declare
fun {Fact N}
   if N=<1 then 1 else N*{Fact N-1} end
end

{Pickle.save Fact "~pvr/public_html/fact"}
```

Since the function is in a `public_html` directory, anyone can load it by giving a URL that specifies the file:

```
declare
Fact={Pickle.load "http://www.info.ucl.ac.be/~pvr/fact"}

{Browse {Fact 10}}
```

Anything stateless can be saved in a pickle, including functions, procedures, classes, functors, records, and atoms. Stateful entities, such as objects and variables, cannot be pickled.

3.1.3 Remote computations and functors

An application can start a computation on a remote host that uses the resources of that host and that continues to interact with the application. The computation is specified as a *functor*, which is the standard way to define computations with the resources they need. A functor is a module specification that makes explicit the resources that the module needs (see Section 2.3).

First we create a new Mozart process that is ready to accept new computations:

```
declare
R={New Remote.manager init(host:"rainbow.info.ucl.ac.be")}
```

Let's make the process do some work. We define a functor that does the work when we evaluate it:

```
declare F M
F=functor export x:X define X={Fact 30} end

M={R apply(F $)}

{Browse M.x}
```

The result `X` is returned to the client site in the module `M`, which is calculated on the remote site and returned to the application site. The module is a record and the result is at the field `x`, namely `M.x`. The module should not reference any resources. If it does, an exception will be raised in the thread doing the `apply`.

Any Oz statement *S* can be executed remotely by creating a functor:

```
F=functor import ResourceList export Results define S end
```

To evaluate this functor remotely, the client executes `M={R apply(F $)}`. The *ResourceList* must list all the resources used by *S*. If not all are listed then an exception will be raised in the thread doing the `apply`. The remote execution will use the resources of the remote site and return a module `M` that contains all the fields mentioned in *Results*. If *S* does not use any resources, then there is a slightly simpler way to do remote computations. The next section shows how by building a simple compute server.

A second solution is to use a functor with an external reference:

```
declare F M X in
F=functor define {Fact 30 X} end

M={R apply(F $)}
{Browse X}
```

This functor is not stateless, but it's all right since we are not pickling the functor. In fact, it's quite possible for functors to have external references. Such functors are called *computed functors*. They can only be pickled if the external references are to stateless entities.

A third solution is for the functor itself to install the compute server on the remote site. This is a more general solution: it *separates* the distribution aspect (setting up the remote site to do the right thing) from the particular computations that we want to do. We give this solution later in the tutorial.

3.2 Servers

A server is a long-lived computation that provides a service to clients. We will show progressively how to build different kinds of servers.

3.2.1 The hello server

Let's build a basic server that returns the string `"Hello world"` to clients. The first step is to create the server. Let's do this and also make the server available through a URL.

```
% Create server
declare Str Prt Srv in
{NewPort Str Prt}
thread
   {ForAll Str proc {$ S} S="Hello world" end}
end
proc {Srv X}
   {Send Prt X}
end

% Make server available through a URL:
% (by using a filename that is also accessible by URL)
{Pickle.save {Connection.offerUnlimited Srv}
             "/usr/staff/pvr/public_html/hw"}
```

All the above must be executed on the server site. Later on we will show how a client can create a server remotely.

Any client that knows the URL can access the server:

```
declare Srv in
Srv={Connection.take {Pickle.load "http://www.info.ucl.ac.be/~pvr/hw"}}

local X in
   {Srv X}
   {Browse X}
end
```

This will show `"Hello world"` in the browser window.

By taking the connection, the client gets a reference to the server. This conceptually merges the client and server computation spaces into a single computation space. The client and server can then communicate as if they were in the same process. Later on, when the client forgets the server reference, the computation spaces become separate again.

3.2.2 The hello server with stationary objects

The previous section shows how to build a basic server using a port to collect messages. There is in fact a much simpler way, namely by using stationary objects. Here's how to create the server:

```
declare
class HelloWorld
   meth hw(X) X="Hello world" end
end

Srv={NewStat HelloWorld hw(_)} % Requires an initial method
```

The client calls the server as `{Srv hw(X)}`. The class `HelloWorld` can be replaced by any class. The only difference between this and creating a centralized object is that `New` is replaced by `NewStat`. This specifies the distributed semantics of the object independently of the object's class.

3.2.3 Making stationary objects

Stationary entities are a very important abstraction. Mozart provides two operations to make entities stationary. The first is creating a stationary object:

```
declare
Object={NewStat Class Init}
```

When executed on a site, the procedure `NewStat` takes a class and an initial message and creates an object that is stationary on that site. We define `NewStat` as follows.

16a ⟨Stationary object 16a⟩≡

```
declare
⟨MakeStat definition 16b⟩

proc {NewStat Class Init Object}
   Object={MakeStat {New Class Init}}
end
```

NewStat is defined in terms of MakeStat. The procedure MakeStat takes an object or a one-argument procedure and returns a one-argument procedure that obeys exactly the same language semantics and is stationary. We define {MakeStat PO StatP} as follows, where input PO is an object or a one-argument procedure and output StatP is a one-argument procedure. [5]

16b ⟨MakeStat definition 16b⟩≡

```
proc {MakeStat PO ?StatP}
   S P={NewPort S}
   N={NewName}
in
   % Client side:
   proc {StatP M}
   R in
      {Send P M#R}
      if R==N then skip else raise R end end
   end
   % Server side:
   thread
      {ForAll S
       proc {$ M#R}
          thread
             try {PO M} R=N catch X then R=X end
          end
       end}
   end
end
```

StatP preserves exactly the same language semantics as PO. In particular, it has the same concurrency behavior and it raises the same exceptions. The new name N is a globally-unique token. This ensures that there is no conflict with any exceptions raised by ProcOrObj.

3.2.4 A compute server

One of the promises of distributed computing is making computations go faster by exploiting the parallelism inherent in networks of computers. A first step is to create a compute server, that is, a server that accepts any computation and uses its computational resources to do the computation. Here's one way to create a compute server:

[5] One-argument procedures are not exactly objects, since they do not have features. For all practical purposes not requiring features, though, one-argument procedures and objects are interchangeable.

```
declare
class ComputeServer
   meth init skip end
   meth run(P) {P} end
end

C={NewStat ComputeServer init}
```

The compute server can be made available through a URL as shown before. Here's how a client uses the compute server:

```
declare
fun {Fibo N}
   if N<2 then 1 else {Fibo N-1}+{Fibo N-2} end
end

% Do first computation remotely
local F in
   {C run(proc {$} F={Fibo 30} end)}
   {Browse F}
end

% Do second computation locally
local F in
   F={Fibo 30}
   {Browse F}
end
```

This first does the computation remotely and then repeats it locally. In the remote case, the variable `F` is shared between the client and server. When the server binds it, its value is immediately sent to the server. This is how the client gets a result from the server.

Any Oz statement *S* that does not use resources can be executed remotely by making a procedure out of it:

```
P=proc {$} S end
```

To run this, the client just executes `{C run(P)}`. Because Mozart is fully network-transparent, *S* can be any statement in the language: for example, *S* can define new classes inheriting from client classes. If *S* uses resources, then it can be executed remotely by means of functors. This is shown in the previous section.

3.2.5 A compute server with functors

The solution of the previous section is reasonable when the client and server are independent computations that connect. Let's now see how the client itself can start up a compute server on a remote site. The client first creates a new Mozart process:

```
declare
R={New Remote.manager init(host:"rainbow.info.ucl.ac.be")}
```

Then the client sends a functor to this process that, when evaluated, creates a compute server:

```
declare F C
F=functor
  export cs:CS
  define
     class ComputeServer
        meth init skip end
        meth run(P) {P} end
     end
     CS={NewStat ComputeServer init}
  end

C={R apply(F $)}.cs  % Set up the compute server
```

The client can use the compute server as before:

```
local F in
   {C run(proc {$} F={Fibo 30} end)}
   {Browse F}
end
```

3.2.6 A dynamically-extensible server

Sometimes a server has to be upgraded, for example to add extra functionality or to fix a bug. We show how to upgrade a server without stopping it. This cannot be done in Java. In Mozart, the upgrade can even be done interactively. A person sits down at a terminal anywhere in the world, starts up an interactive Mozart session, and upgrades the server while it is running.

Let's first define a generic upgradable server:

```
declare
proc {NewUpgradableStat Class Init ?Upg ?Srv}
   Obj={New Class Init}
   C={NewCell Obj}
in
   Srv={MakeStat
          proc {$ M} {@C M} end}
   Upg={MakeStat
          proc {$ Class2#Init2} C := {New Class2 Init2} end}
end
```

This definition must be executed on the server site. It returns a server Srv and a stationary procedure Upg used for upgrading the server. The server is upgradable because it does all object calls indirectly through the cell C.

A client creates an upgradable compute server almost exactly as it creates a fixed compute server, by executing the following on the server site:

```
declare Srv Upg in
Srv={NewUpgradableStat ComputeServer init Upg}
```

Let's now upgrade the compute server while it is running. We first define a new class `CComputeServer` and then we upgrade the server with an object of the new class:

```
declare
class CComputeServer from ComputeServer
   meth run(P Prio<=medium)
      thread
         {Thread.setThisPriority Prio}
         ComputeServer,run(P)
      end
   end
end

Srv2={Upg CComputeServer#init}
```

That's all there is to it. The upgraded compute server overrides the `run` method with a new method that has a default. The new method supports the original call `run(P)` and adds a new call `run(P Prio)`, where `Prio` sets the priority of the thread doing computation `P`.

The compute server can be upgraded indefinitely since garbage collection will remove any unused old compute server code. For example, it would be nice if the client could find out how many active computations there are on the compute server before deciding whether or not to do a computation there. We leave it to the reader to upgrade the server to add a new method that returns the number of active computations at each priority level.

3.3 Practical tips

This section gives some practical programming tips to improve the network performance of distributed applications: timing and memory problems, avoiding sending data that is not used at the destination and avoiding sending classes when sending objects across the network.

3.3.1 Timing and memory problems

When the distribution structure of an application is changed, then one must be careful not to cause timing and memory problems.

- When a reference `X` is exported from a site (i.e., put in a message and sent) and `X` refers directly or indirectly to unused modules then the modules will be loaded into memory. This is so even if they will never be used.

- Relative timings between different parts of a program depend on the distribution structure. For example, unsynchronized producer/consumer threads may work fine if both are on the same site: it suffices to give the producer thread a slightly lower priority. If the threads are on different sites, the producer may run faster and cause a memory leak.

- If the same record is sent repeatedly to a site, then a new copy of the record will be created there each time. This is true because records don't have global names. The lack of global names makes it faster to send records across the network.

3.3.2 Avoiding sending useless data

When sending a procedure over the network, be sure that it doesn't contain calculations that could have been done on the original site. For example, the following code sends the procedure `P` to remote object `D`:

```
declare
R={MakeTuple big 100000}  % A very, very big tuple
proc {P X} X=R.2710 end   % Procedure that uses tuple field 2710
{D addentry(P)}           % Send P to D, where it is executed
```

If `D` executes `P`, then the big tuple `R` is transferred to `D`'s site, where field number `2710` is extracted. With 100,000 fields, this means 400KB is sent over the network! Much better is to extract the field before sending `P`:

```
declare
R={MakeTuple big 100000}
F=R.2710                  % Extract field 2710 before sending
proc {P X} X=F end
{D addentry(P)}
```

This avoids sending the tuple across the network. This technique is a kind of partial evaluation. It is useful for almost any Oz entity, for example procedures, functions, classes, and functors.

3.3.3 Avoiding sending classes

When sending an object across the network, it is good to make sure that the object's class exists at the destination site. This avoids sending the class code across the network. Let's see how this works in the case of a collaborative tool. Two sites have identical binaries of this tool, which they are running. The two sites send objects back and forth. Here's how to write the application:

```
declare
class C
   % ... lots of class code comes here
end
functor
define
```

```
      Obj={New C init}
      % ... code for the collaborative tool
   end
```

This creates the class C for the functor to reference. This means that all copies of the binary with this functor will reference the *same* class, so that an object arriving at a site will recognize the *same* class as its class on the original site.

Here's how *not* to write the application:

```
   functor
   define
      class C
         % ... lots of class code comes here
      end
      Obj={New C init}
      % ... code for the collaborative tool
   end
```

Do you see why? Think first before reading the next paragraph! For a hint read Section 2.1.4.

In both solutions, the functor is applied when the application starts up. In the second solution, this defines a new and different class C on each site. If an object of class C is passed to a site, then the site will ask for the class code to be passed too. This can be very slow if the class is big–for TransDraw it makes a difference of several minutes on a typical Internet connection. In the first solution, the class C is defined *before* the functor is applied. When the functor is applied, the class already exists! This means that all sites have exactly the same class, which is part of the binary on each site. Objects passed between the sites will never cause class code to be sent.

4

Failure Model

Distributed systems have the partial failure property, that is, part of the system can fail while the rest continues to work. Partial failures are not at all rare. Properly-designed applications must take them into account. This is both good and bad for application design. The bad part is that it makes applications more complex. The good part is that applications can take advantage of the redundancy offered by distributed systems to become more robust.

The Mozart failure model defines what failures are recognized by the system and how they are reflected in the language. The system recognizes permanent site failures that are instantaneous and both temporary and permanent communication failures. The permanent site failure mode is more generally known as fail-silent with failure detection, that is, a site stops working instantaneously, does not communicate with other sites from that point onwards, and the stop can be detected from the outside. The system provides mechanisms to program with language entities that are subject to failures.

The Mozart failure model is accessed through the module `Fault`[1]. This chapter explains and justifies this functionality, and gives examples showing how to use it. We present the failure model in two steps: the basic model and the advanced model. To start writing fault-tolerant applications it is enough to understand the basic model. To build fault-tolerant abstractions it is often necessary to use the advanced model.

In its current state, the Mozart system provides only the primitive operations needed to detect failure and reflect it in the language. The design and implementation of fault-tolerant abstractions within the language by using these primitives is the subject of ongoing research. This chapter and the next one give the first results of this research. All comments and suggestions for improvements are welcome.

4.1 Fault states

All failure modes are defined with respect to both a language entity and a particular site. For example, one would like to send a message to a port from a given site. The site may or may not be able to send the message. A language entity can be in three fault states on a given site:

- The entity works normally (local fault state `ok`).

[1] Chapter *Detecting and Handling Distribution Problems:* `Fault`, *(System Modules)*

- The entity is temporarily not working (local fault state tempFail). This is because a remote site crucial to the entity is currently unreachable due to a network problem. This fault state can go away. A limitation of the current release is that temporary problems are indicated only after a long delay time.
- The entity is permanently not working (local fault state permFail). This is because a site crucial to the entity has crashed. This fault state is permanent.

If the entity is currently not working, then it is guaranteed that the fault state will be either tempFail or permFail. The system cannot always determine whether a fault is temporary or permanent. In particular, a tempFail may hide a site crash. However, network failures can always be considered temporary since the system actively tries to reestablish another connection.

4.1.1 Temporary faults

The fault state tempFail exists to allow the application to react quickly to temporary network problems. It is raised by the system as soon as a network problem is recognized. It is therefore fundamentally different from a time-out. For example, TCP gives a time-out after some minutes. This duration has been chosen to be very long, approximating infinity from the viewpoint of the network connection. After the time-out, one can be sure that the connection is no longer working.

The purpose of tempFail is quite different from a time-out. It is to *inform* the application of network problems, not to mark the *end* of a connection. For example, an application might be connected to a given server. If there are problems with this server, the application would like to be informed quickly so that it can try connecting to another server. A tempFail fault state will therefore be relatively frequent, much more frequent than a time-out. In most cases, a tempFail fault state will eventually go away.

It is possible for a tempFail state to last forever. For example, if a user disconnects the network connection of a laptop machine, then only he or she knows whether the problem is permanent. The application cannot in general know this. The decision whether to continue waiting or to stop the wait can cut through all levels of abstraction to appear at the top level (i.e., the user). The application might then pop up a window to ask the user whether to continue waiting or not. The important thing is that the network layer does not make this decision; the application is completely free to decide or to let the user decide.

4.1.2 Remote problems

The local fault states ok, tempFail, and permFail say whether an entity operation can be performed locally. An entity can also contain information about the fault states on other sites. For example, say the current site is waiting for a variable binding, but the remote site that will do the binding has crashed. The current site can find this out. The following remote problems are identified:

- At least one of the other sites referencing the entity can no longer perform operations on the entity (fault state remoteProblem(permSome)). The sites may or may not have crashed.

- All of the other sites referencing the entity can no longer perform operations on the entity (fault state `remoteProblem(permAll)`). The sites may or may not have crashed.
- At least one of the other sites referencing the entity is currently unreachable (fault state `remoteProblem(tempSome)`).
- All of the other sites referencing the entity are currently unreachable (fault state `remoteProblem(tempAll)`).

All of these cases are identified by the fault state `remoteProblem(I)`, where the argument `I` identifies the problem. A permanent remote problem never goes away. A temporary remote problem can go away, just like a `tempFail`.

Even if there exists a remote problem, it is not always possible to return a `remoteProblem` fault state. This happens if there are problems with a proxy that the owner site does not know about. This also happens if the owner site is inaccessible. In that case it might not be possible to learn anything about the remote sites.

The complete fault state of an entity consists of at most one element from the set {`tempFail`, `permFail`} together with at most two elements from the set {`remoteProblem(permSome)`, `remoteProblem(permAll)`, `remoteProblem(tempSome)`, `remoteProblem(tempAll)`}. Permanent remote problems mask temporary ones, i.e., if `remoteProblem(permSome)` is detected then `remoteProblem(tempSome)` cannot be detected. If a (temporary or permanent) problem exists on *all* remote sites (e.g., `remoteProblem(permAll)`) then the problem also exists on *some* sites (e.g., `remoteProblem(permSome)`).

4.2 Basic model

We present the failure model in two steps: the basic model and the advanced model. The simplest way to start writing fault-tolerant applications is to use the basic model. The basic model allows to enable or disable synchronous *exceptions* on language entities. That is, attempting to perform operations on entities with faults will either block or raise an exception without doing the operation. The fault detection can be enabled separately on each of two levels: a per-site level or a per-thread level (see Section 4.2.4).

Exceptions can be enabled on logic variables, ports, objects, cells, and locks. All other entities, e.g., records, procedures, classes, and functors, will never raise an exception since they have no fault states (see Section 4.4.1). Attempting to enable an exception on such an entity is allowed but has no observable effect.

The advanced model allows to install or deinstall *handlers* and *watchers* on entities. These are procedures that are invoked when there is a failure. Handlers are invoked synchronously (when attempting to perform an operation) and watchers are invoked asynchronously (in their own thread as soon as the fault state is known). The advanced model is explained in Section 4.3.

4.2.1 Enabling exceptions on faults

By default, new entities are set up so that an exception will be raised on fault states `tempFail` or `permFail`. The following operations are provided to do other kinds of fault detection:

`fun {Fault.defaultEnable FStates}`

sets the site's default for detected fault states to `FStates`. Each site has a default that is set independently of that of other sites. Enabling site or thread-level detection for an entity overrides this default. Attempting to perform an operation on an entity with a fault state in the default `FStates` raises an exception. The `FStates` can be changed as often as desired. When the system starts up, the defaults are set up as if the call `{Fault.defaultEnable [tempFail permFail]}` had been done.

`fun {Fault.defaultDisable}`

disables the default fault detection. This function is included for symmetry. It is exactly the same as doing `{Fault.defaultEnable nil}`.

`fun {Fault.enable Entity Level FStates}`

is a more targeted way to do fault detection. It enables fault detection on a given entity at a given level. If a fault in `FStates` occurs while attempting an operation at the given level, then an exception is raised instead of doing the operation. The `Entity` is a reference to any language entity. Exceptions are enabled only if the entity is an object, cell, port, lock, or logic variable. The `Level` is `site`, `'thread'(this)` (for the current thread), or `'thread'(T)` (for an arbitrary thread identifier `T`).[2] More information on levels is given in Section 4.2.4.

`fun {Fault.disable Entity Level}`

disables fault detection on the given entity at the given level. If a fault occurs, then the system does nothing at the given level, but checks whether any exceptions are enabled at the next higher level. This is *not* the same as `{Fault.enable Entity Level nil}`, which always causes the entity to block at the given level.

The function `Fault.enable` returns `true` if and only if the enable was successful, i.e., the entity was not already enabled for that level. The function `Fault.disable` returns `true` if and only if the disable was successful, i.e., the entity was already enabled for that level. The functions `Fault.defaultEnable` and `Fault.defaultDisable` always return `true`. At its creation, an entity is not enabled at any level. All four functions raise a type error exception if their arguments do not have the correct type.

4.2.2 Binding logic variables

A logic variable can be declared before it is bound. What happens to its enabled exceptions when it is bound? For example, let's say variable `V` is enabled with `FS_v` and port `P` is enabled with `FS_p`. What happens after the binding `V=P`? In this case, the binding gives `P`, which keeps the enabled exceptions `FS_p`. The enabled exceptions `FS_v` are discarded.

The following cases are possible. We assume that variable `V` is enabled with fault detection on fault states `FS_v`.

- `V` is bound to a nonvariable entity `X` that has no enabled exceptions. In this case, the enabled exceptions `FS_v` are transferred to `X`.

[2] Since `thread` is already used as a keyword in the language, it has to be quoted to make it an atom.

- V is bound to a nonvariable entity X that already has enabled exceptions FS_x. In this case, X keeps its enabled exceptions and FS_v is discarded.
- V is bound to another logic variable W that might have enabled exceptions. In this case, the resulting variable keeps *one* set of enabled exceptions, either FS_v or FS_w (if the latter exists). Which one is not specified.

These cases follow from three basic principles:

- A logic variable that is "observed", e.g., it has fault detection with enabled exceptions, will be "observed" at all instants of time. That is, it will keep some kind of fault detection even after it is bound.
- A nonvariable entity is never bothered by being bound to a variable. That is, the nonvariable's fault detection (if there is any) can only be modified by explicit commands from Fault, never from being bound to a variable.
- Any language entity that is set up with a set of enabled exceptions will have exactly *one* set of enabled exceptions, even if it is bound. There is no attempt to "combine" the two sets.

4.2.3 Exception formats

The exceptions raised have the format

```
system(dp(entity:E conditions:FS op:OP) ...)
```

where the four arguments are defined as follows:

- E is the entity on which the operation was attempted. A temporary limitation of the current release is that if the entity is an object, then E is undefined.
- FS is the list of actual fault states occurring at the site on which the operation was attempted. This list is a subset of the list for which fault detection was enabled. Each fault state in FS may have an extra field info that gives additional information about the fault. The possible elements of FS are currently the following:
  - tempFail(info:I) and permFail(info:I), where I is in {state, owner}. The info field only exists for objects, cells, and locks.
  - remoteProblem(tempSome), remoteProblem(permSome), remoteProblem(tempAll), and remoteProblem(permAll).
- OP indicates which attempted operation caused the exception. The possible values of OP are currently:
  - For logic variables: bind(T), wait, and isDet, where T is what the variable was attempted to be bound with.
  - For cells: cellExchange(Old New), cellAssign(New), and cellAccess(Old), where Old is the cell content before the attempted operation and New is the cell content after the attempted operation.

- For locks: `'lock'`.[3]
- For ports: `send(Msg)`, where `Msg` is the message attempted to be sent to the port.
- For objects: `objectExchange(Attr Old New)`, `objectAssign(Attr New)`, `objectAccess(Attr Old)`, and `objectFetch`, where `Attr` is the name of the object attribute, `Old` is the attribute value before the attempted operation, and `New` is the attribute value after the attempted operation. A limitation of the current release is that the attempted operation cannot be retried. The `objectFetch` operation exists because object-records are copied lazily: the first time the object is used, the object-record is fetched over the network, which might fail.

4.2.4 Levels of fault detection

There are three levels of fault detection, namely default site-based, site-based, and thread-based. A more specific level, if it exists, overrides a more general level. The most general is *default site-based*, which determines what exceptions are raised if the entity is not enabled at the site or thread level. Next is *site-based*, which detects a fault for a specific entity when an operation is tried on one particular site. Finally, the most fine-grained is *thread-based*, which detects a fault for a specific entity when an operation is tried in a particular thread.

The site-based and thread-based levels have to be enabled specifically for a given entity. The function `{Fault.enable Entity Level FStates}` is used, where `Level` is either `site` or `'thread'(T)`. The thread `T` is either the atom `this` (which means the current thread), or a thread identifier. Any thread's identifier can be obtained by executing `{Thread.this T}` in the thread.

The thread-based level is the most specific; if it is enabled it overrides the two others in its thread. The site-based level, if it is enabled, overrides the default. If neither a thread-based nor a site-based level are enabled, then the default is used. Even if the actual fault state does not give an exception, the mere fact that a level is enabled always overrides the next higher level.

For example, assume that the cell `C` is on a site with default detection for `[tempFail permFail]` and thread-based detection for `[permFail]` in thread `T1`. What happens if many threads try to do an exchange if C's actual fault state is `tempFail`? Then thread `T1` will block, since it is set up to detect only `permFail`. All other threads will raise the exception `tempFail`, since the default covers it and there is no enable at the site or thread levels. Thread `T1` will continue the operation when and if the `tempFail` state goes away.

4.2.5 Levels and sitedness

The `Fault` module has both sited and unsited operations. Both setting the default and enabling at the site level are *sited*. This protects the site from remote attempts to change its settings. Enabling at the thread level is *unsited*. This allows fault-tolerant abstractions to be network-transparent, i.e., when passed to another site they continue to work.

[3] Since **`lock`** is already used as a keyword in the language, it has to be quoted to make it an atom.

To be precise, the calls `{Fault.enable E site ...}` and `{Fault.install E site ...}`, will only work on the home site of the `Fault` module. A procedure containing these calls may be passed around the network at will, and executed anywhere. However, any attempt to execute either call on a site different from the `Fault` module's home site will raise an exception.[4] The calls `{Fault.enable E 'thread'(T) ...}` and `{Fault.install E 'thread'(T) ...}` will work anywhere. A procedure containing these calls may be passed around the network at will, and will work correctly anywhere. Of course, since threads are sited, `T` has to identify a thread on the site where the procedure is executed.

4.3 Advanced model

The basic model lets you set up the system to raise an exception when an operation is attempted on a faulty entity. The advanced model extends this to *call a user-defined procedure*. Furthermore, the advanced model can call the procedure *synchronously*, i.e., when an operation is attempted, or *asynchronously*, i.e., as soon as the fault is known, even if no operation is attempted. In the synchronous case, the procedure is called a *fault handler*, or just *handler*. In the asynchronous case, the procedure is called *watcher*.

4.3.1 Lazy detection with handlers

When an operation is attempted on an entity with a problem, then a handler call replaces the operation. This call is done in the context of the thread that attempted the operation. If the entity works again later (which is possible with `tempFail` and `remoteProblem`) then the handler can just try the operation again.

In an exact analogy to the basic model, a fault handler can be installed on a given entity at a given level for a given set of fault states. The possible entities, levels, and fault states are exactly the same. What happens to handlers on logic variables when the variables are bound is exactly the same as what happens to enabled exceptions in Section 4.2.2. For example, when a variable with handler `H_v1` is bound to another variable with handler `H_v2`, then the result has exactly one handler, say `H_v2`. The other handler `H_v1` is discarded. When a variable with handler is bound to a port with handler, then the port's handler survives and the variable's handler is discarded.

Handlers are installed and deinstalled with the following two built-in operations:

`fun {Fault.install Entity Level FStates HandlerProc}`

installs handler `HandlerProc` on `Entity` at `Level` for fault states `FStates`. If an operation is attempted and there is a fault in `FStates`, then the operation is replaced by a call to `HandlerProc`. At most one handler can be installed on a given entity at a given level.

`fun {Fault.deInstall Entity Level}`

deinstalls a previously installed handler from `Entity` at `Level`.

[4] Note that each site has its own `Fault` module.

The function `Fault.install` returns **`true`** if and only if the installation was successful, i.e., the entity did not already have an installation or an enable for that level. The function `Fault.deInstall` returns **`true`** if and only if the deinstall was successful, i.e., the entity had a handler installed for that level. Both functions raise a type error exception if their arguments do not have the correct type.

A handler `HandlerProc` is a three-argument procedure that is called as `{HandlerProc E FS OP}`. The arguments `E`, `FS`, and `OP`, are exactly the same as in a distribution exception.

A modification of the current release with respect to handler semantics is that handlers installed on *variables* always retry the operation after they return.

4.3.2 Eager detection with watchers

Fault handlers detect failure synchronously, i.e., when an operation is attempted. One often wants to be informed earlier. The advanced model allows the application to be informed asynchronously and eagerly, that is, as soon as the site finds out about the failure. Two operations are provided:

`fun {Fault.installWatcher Entity FStates WatcherProc}`

installs watcher `WatcherProc` on `Entity` for fault states `FStates`. If a fault in `FStates` is detected on the current site, then `WatcherProc` is invoked in its own new thread. A watcher is automatically deinstalled when it is invoked. Any number of watchers can be installed on an entity. The function always returns **`true`**, since it is always possible to install a watcher.

`fun {Fault.deInstallWatcher Entity WatcherProc}`

deinstalls (i.e., removes) one instance of the given watcher from the entity on the current site. If no instance of `WatcherProc` is there to deinstall, then the function returns **`false`**. Otherwise, it returns **`true`**.

A watcher `WatcherProc` is a two-argument procedure that is called as `{WatcherProc E FS}`. The arguments `E` and `FS` are exactly the same as in a distribution exception or in a handler call.

4.4 Fault states for language entities

This section explains the possible fault states of each language entity in terms of its distributed semantics. The fault state is a consequence of two things: the entity's distributed implementation and the system's failure mode. For example, let's consider a variable. There is one owner site and a set of proxy sites. If a variable proxy is on a crashed site and the owner site is still working, then to another variable proxy this will be a `remoteProblem`. If the owner site crashes, then all proxies will see a `permFail`.

4.4.1 Eager stateless data

Eager stateless data, namely records, procedures, functions, classes, and functors, are copied immediately in messages. There are no remote references to eager stateless data, which are always local to a site. So their only possible fault state is `ok`.

In future releases, procedures, functions, and functors will not send their code immediately in the message, but will send only their global name. Upon arrival, if the code is not present, then it will be immediately requested. This will guarantee that code is sent at most once to a given site. This will introduce fault states `tempFail` and `permFail` if the site containing the code becomes unreachable or crashes.

4.4.2 Sited entities

Sited entities can be referenced remotely but can only be used on their home site. Attempting to use one outside of its home site immediately raises an exception. Detecting this does not need any network operations. So their only possible fault state is `ok`.

4.4.3 Ports

A port has one owner site and a set of proxy sites. The following failure modes are possible:

- Normal operation (`ok`).
- Owner site down (`permFail` and `remoteProblem(I)` where `I` is both `permSome` and `permAll`).
- Owner site unreachable (`tempFail`).

A port has a single operation, `Send`, which can complete if the fault state is `ok`. The `Send` operation is asynchronous, that is, it completes immediately on the sender site and at some later point in time it will complete on the port's owner site. The fact that it completes on the sender site does not imply that it will complete on the owner site. This is because the owner site may fail.

4.4.4 Logic variables

A logic variable has one owner site and a set of proxy sites. The following failure modes are possible:

- Normal operation (`ok`).
- Owner site down (`permFail` and `remoteProblem(I)` where `I` is both `permSome` and `permAll`).
- Owner site unreachable (`tempFail`).
- Some or all proxy sites down (`remoteProblem(I)` where `I` is both `permSome` and `permAll`).
- Some or all proxy sites unreachable (`remoteProblem(tempSome))`). It is impossible to have `remoteProblem(tempAll)` in the current implementation.

A logic variable has two operations, binding and waiting until bound. Bind operations are explicit in the program text. Most wait operations are implicit since threads block until their data is available. The bind operation will only complete if the fault state is `ok` or `remoteProblem`.

If the application binds a variable, then its wait operation is only guaranteed to complete if the fault state is `ok`. When it completes, this means that another proxy has bound the variable. If the fault state is `remoteProblem`, then the wait operation may not be able to complete if the problem exists at the proxy that was supposed to bind the variable. This is *not* a `tempFail` or `permFail`, since a third proxy can successfully bind the variable. But from the application's viewpoint, it may still be important to know about this problem. Therefore, the fault state `remoteProblem` is important for variables.

A common case for variables is the client-server. The client sends a request containing a variable to the server. The server binds the variable to the answer. The variable exists only on the client and server sites. In this case, if the client detects a `remoteProblem` then it knows that the variable binding will be delayed or never done.

4.4.5 Cells and locks

Cells and locks have almost the same failure behavior. A cell or lock has one owner site and a set of proxy sites. At any given time instant, the cell's state pointer or the lock's token is at one proxy or in the network. The following failure modes are possible:

- Normal operation (`ok`).
- State pointer not present and owner site down (`permFail(info:owner)` and `remoteProblem(permSome)`).
- State pointer not present and owner site unreachable (`tempFail(info:owner)`).
- State pointer lost and owner site up (`permFail(info:state)`, `remoteProblem(permAll)`, and `remoteProblem(permSome)`). This failure mode is only possible for cells. If a lock token is lost then the owner recreates it.
- State pointer unreachable and owner site up (`tempFail(info:state)`).
- State pointer present and owner site down (`remoteProblem(permAll)` and `remoteProblem(per`
- State pointer present and owner site unreachable (`remoteProblem(tempAll)` and `remoteProblem(tempSome)`).

A cell has one primitive operation, a state update, which is called `Exchange`. A lock has two implicit operations, acquiring the lock token and releasing it. Both are implemented by the same distributed protocol.

4.4.6 Objects

An object consists of an object-record that is a lazy chunk and that references the object's features, a cell, and a class. The object-record is lazy: it is copied to the site when the object is used for the first time. This means that the following failure modes are possible:

- Normal operation (`ok`).
- Object-record or state pointer not present and owner site down (`permFail(info:owner)` and `remoteProblem(permSome)`).
- Object-record or state pointer not present and owner site unreachable (`tempFail(info:owner)`).
- State pointer lost and owner site up (`permFail(info:state)`, `remoteProblem(permAll)`, and `remoteProblem(permSome)`).
- State pointer unreachable and owner site up (`tempFail(info:state)`).
- Object-record and state pointer present and owner site down (`remoteProblem(permAll)` and `remoteProblem(permSome)`).
- Object-record and state pointer present and owner site unreachable (`remoteProblem(tempAll)` and `remoteProblem(tempSome)`).

Compared to cells, objects have two new failure modes: the object-record can be temporarily or permanently absent. In both cases the object cannot be used, so we simply consider the new failure modes to be instances of `tempFail` and `permFail`.

5

Limitations and Modifications

The current release has the following limitations and modifications with respect to the specifications of the distribution model and the failure model. A *limitation* is an operation that is specified but is not possible or has lower performance in the current release. A *modification* is an operation that is specified but behaves differently in the current release.

Most of the limitations and modifications listed here will be removed in future releases.

5.1 Performance limitations

These reduce performance but do not affect language semantics. They can safely be ignored if performance is not an issue.

- The following problems are related to the `Remote` module and virtual sites (see also Chapter *Spawning Computations Remotely:* `Remote`, *(System Modules)*).
  - On some platforms (in particular, Solaris), the operating system in its default configuration does not support virtual sites efficiently (see also Chapter *Spawning Computations Remotely:* `Remote`, *(System Modules)*). This is due to a system-wide limit on the number of shared memory pages. For Solaris, the default is six shared pages per process and 100 system-wide. Changing this limit requires rebooting the machine. Since at least two pages are needed for efficient communication, the default value results in poor performance if a site connects to more than three virtual sites.
  - The Mozart system does its best to reclaim shared memory identifiers, even upon process crashes, but it is still possible that some shared memory pages become unaccounted for and thus stay forever in the operating system. If this happens please use Unix utilities to get rid of them. On Solaris and Linux there are two, namely `ipcs` and `ipcrm`.
- The code of functions, procedures, classes, and functors (but not objects) is always inserted in messages, even if the code is already present at the destination. In future releases, the code will be copied across the network only if it is not present on the destination site. In both current and future releases, at most a single copy of the code can exist per site.

- The distributed garbage collection algorithm reclaims all unused entities except those forming a reference cycle that exists on at least two different owner sites (a *cross-site cycle*). For example, if two sites each own an object that references the other, then they will never be reclaimed. It is up to the programmer to break the cycle by updating one of the objects to no longer reference the other.
- If a site crashes that has references to entities created on other sites, then these entities are not garbage-collected. Future releases will incorporate a lease-based or similar technique to recover such entities.
- The fault state `tempFail` is indicated only after a long delay. In future releases, the delay will be very short and based on adaptive observation of actual network behavior.

5.2 Functionality limitations

These affect what operations are available to the programmer. They document where the full language specification is not implemented. We hope that the undergrowth of limitations is sparse enough to let the flowers of Oz grow unwithered.[1]

- On Windows, the `Remote` module has limited functionality. Only a single option is possible for `fork`, namely `sh`. Future releases will add more options.
- The `Connection` module does not work correctly for applications separated by a firewall. This limitation will be addressed in a future release.
- Threads, dictionaries, arrays, and spaces are sited, even though they are in base modules. In future releases, it is likely that dictionaries and arrays will be made unsited. Threads and spaces will be made stationary entities that can be called remotely (like ports).
- When a reference to a constrained variable (finite domain, finite set, or free record) is passed to another site, then this reference is converted to a *future*. The future will be bound when the constrained variable becomes determined.
- If an exception is raised or a handler or watcher is invoked for an *object*, then the `Entity` argument is undefined. For handlers and watchers, this limitation can be bypassed by giving the handler and watcher procedures a reference to the object.
- If an exception is raised or a handler is invoked for an *object*, then the attempted object operation cannot be retried. This limitation can be bypassed by programming the object so that it is known where in which method the error was detected.

5.3 Modifications

There is currently only one modification.

- A handler installed on a *variable* will retry the operation (i.e., bind or wait) after it returns. That is, the handler is inserted before the operation instead of replacing the operation.

[1] C. A. R. Hoare, *The Emperor's Old Clothes*, 1980 Turing Award Lecture.

Bibliography

[1] Vijay Saraswat. *Concurrent Constraint Programming*. MIT Press, 1994.

Index