

Distributed Programming in Mozart - A Tutorial Introduction

**Peter Van Roy
Seif Haridi
Per Brand**

**Version 1.2.3
December 1, 2001**



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 still incomplete and subject to change from day to day.

Credits

Mozart logo by Christian Lindig

License Agreement

This software and its documentation are copyrighted by the German Research Center for Artificial Intelligence (DFKI), the Swedish Institute of Computer Science (SICS), and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE AND ITS DOCUMENTATION ARE PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Contents

1	Introduction	1
2	Distribution Model	3
2.1	Language entities	3
2.1.1	Objects	3
2.1.2	Other stateful entities	4
2.1.3	Single-assignment entities	5
2.1.4	Stateless entities	6
2.1.5	Sited entities	7
2.2	Sites	8
2.2.1	Controlled system shutdown	8
2.2.2	Distributed memory management	8
2.3	Bringing it all together	8
3	Basic Operations and Examples	11
3.1	Global naming	11
3.1.1	Connecting applications by means of tickets	12
3.1.2	Persistent data structures by means of pickles	12
3.1.3	Remote computations and functors	13
3.2	Servers	14
3.2.1	The hello server	14
3.2.2	The hello server with stationary objects	15
3.2.3	Making stationary objects	15
3.2.4	A compute server	16
3.2.5	A compute server with functors	17
3.2.6	A dynamically-extensible server	18
3.3	Practical tips	19
3.3.1	Timing and memory problems	19
3.3.2	Avoiding sending useless data	20
3.3.3	Avoiding sending classes	20

4	Mobile Agents	23
4.1	An example agent	24
4.1.1	Installing the agent servers	24
4.1.2	Programming agents	25
4.1.3	There and back again	26
4.1.4	Round and round it goes, where it stops nobody knows	26
4.1.5	Barrier synchronization	26
4.1.6	Definition of <code>AgentServer</code>	26
5	Failure Model	29
5.1	Fault states	29
5.1.1	Temporary faults	30
5.1.2	Remote problems	30
5.2	Basic model	31
5.2.1	Enabling exceptions on faults	31
5.2.2	Binding logic variables	32
5.2.3	Exception formats	33
5.2.4	Levels of fault detection	34
5.2.5	Levels and sitedness	34
5.3	Advanced model	35
5.3.1	Lazy detection with handlers	35
5.3.2	Eager detection with watchers	36
5.4	Fault states for language entities	36
5.4.1	Eager stateless data	36
5.4.2	Sited entities	37
5.4.3	Ports	37
5.4.4	Logic variables	37
5.4.5	Cells and locks	38
5.4.6	Objects	38

6	Fault-Tolerant Examples	41
6.1	A fault-tolerant hello server	41
6.1.1	Definition of <code>SafeSend</code> and <code>SafeWait</code>	42
6.1.2	Definition of <code>FOneOf</code> and <code>FSomeOf</code>	44
6.2	Fault-tolerant stationary objects	44
6.2.1	Using fault-tolerant stationary objects	45
6.2.2	Guard-based fault tolerance	46
6.2.3	Exception-based fault tolerance	48
6.3	A fault-tolerant broadcast channel	50
6.3.1	Sample use (no fault tolerance)	51
6.3.2	Definition (no fault tolerance)	52
6.3.3	Sample use (with fault tolerance)	55
6.3.4	Definition (with fault tolerance)	55
7	Limitations and Modifications	59
7.1	Performance limitations	59
7.2	Functionality limitations	60
7.3	Modifications	60

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.

¹Chapter *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 7).

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.

²Chapter *Spawning Computations Remotely*: `Remote`, (System Modules)

³Chapter *Persistent Values*: `Pickle`, (System Modules)

⁴Chapter *Detecting and Handling Distribution Problems*: `Fault`, (System Modules)

⁵Chapter *Connecting Computations*: `Connection`, (System Modules)

⁶Chapter *Spawning Computations Remotely*: `Remote`, (System Modules)

⁷Chapter *Persistent Values*: `Pickle`, (System Modules)

⁸Chapter *Connecting Computations*: `Connection`, (System Modules)

⁹Chapter *Detecting and Handling Distribution Problems*: `Fault`, (System Modules)

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 (see Section 6.2). 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 5) 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.

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

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

³Chapter *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.

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

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

³Chapter *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:f:x`. 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:

⁴Chapter *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

```

A fault-tolerant version of `NewStat` is given in Section 6.2.2. `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
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:

⁵One-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} {{Access C} M} end}
  Upg={MakeStat
    proc {$ Class2#Init2} {Assign 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.

Mobile Agents

This chapter shows how to program mobile agents in Mozart. We call *agent* any distributed computation that is organized as a set of tasks. A *task* is a computation that uses the resources of a single site. By *resource* we mean the technical definition given in Section 2.1.5, for example, file system, peripherals, networking abilities, operating system access, etc. A task can initiate tasks on other sites, with well-defined specifications of what resources they should use. The distributed behavior of the agent is therefore in first instance decided by the agent itself.

Agents are therefore just resource-aware distributed computations, which can exist on one site or be spread out over more than one site. This definition of agent might seem unnecessarily general, but it is natural in Mozart, where it is as easy for an application to run on one site or a set of sites. For example, here's an agent `A` that concurrently delegates 10 tasks to remote sites and waits until all are done before continuing. The agent servers are represented by `AS0` and `AS9` and the work is represented by functors containing the one-argument procedures `P0` to `P9`. Each procedure binds its argument to the result of its calculation.

```
declare
A=functor
  define
    X0 ... X9
    {AS0 functor define {P0 X0} end}
    ...
    {AS9 functor define {P9 X9} end}
    {Wait X0} ... {Wait X9}
    ...
  end
```

This is efficient. As the distribution model makes clear (see Section 2.1.3), the termination of each remote task is signaled to the original task by exactly one network message, which contains the result of the task's calculation.

The Mozart vision of a universe of agents is a set of fixed places, the agent servers, and a set of evolving computational “webs”, each potentially covering many places at once. A web is what we call an “agent”.

4.1 An example agent

Let's start with a very simple example of an agent that goes somewhere, interrogates the operating system, and then comes back. We show the example in two parts. First, we show how to install agent servers so that the agent has somewhere to go. Then, we will program the agents.

4.1.1 Installing the agent servers

To go to a site, there has to be something at the site that accepts the agent. We call it an *agent server*. An agent server accepts functors (which represent agents or parts of agents) and applies them on its site. To install an agent server on a site, we use the functor `AgentServer` (see Section 4.1.6, below). When `AgentServer` is installed on a site, then it creates an agent server module `AS` on that site along with the following operations:

- The agent server is accessed by `AS.server`. A calculation is started asynchronously on the agent server by invoking `{AS.server F}`, where `F` is a functor that specifies the calculation and the resources it needs.
- The operation `{AS.publishserver FN}`, which when executed creates a file `FN` that contains a ticket to the site's agent server.
- The operation `{AS.getserver UFN ?AS}`, which when inputted an URL or file name `UFN` to any agent server (even on other sites), gives as output a reference to that agent server `AS`.

Let's say that we know an URL `"http://www.info.ucl.ac.be/~pvr/agents.ozf"` that references the functor `AgentServer`. Then the following code creates a local agent server and makes it accessible through the URL `"http://www.info.ucl.ac.be/~pvr/as1"`:

```
declare GetServer in
local
  % Get the AgentServer:
  AgentServer={Pickle.load "http://www.info.ucl.ac.be/~pvr/agents.ozf"}
  % Install AgentServer locally: (this creates an agent server)
  [AS1]={Module.apply [AgentServer]}
  % Publish the agent server:
  {AS1.publishserver "/usr/staff/pvr/public_html/as1"}
in
  GetServer=AS1.getserver
end
```

This code also creates the procedure `GetServer` as defined above.

Let's create a second agent server, a remote one, and make it accessible through the URL `"http://www.info.ucl.ac.be/~pvr/as2"`:

```
local
  RF=functor import Pickle Module export done:D
```

```

define
  AgentServer={Pickle.load "http://www.info.ucl.ac.be/~pvr/agents.ozf"}
  [AS2]={Module.apply [AgentServer]}
  {AS2.publishserver "/usr/staff/pvr/public_html/as2"}
end
RM={New Remote.manager init}
M={RM apply(RF $)}
in
  skip
end

```

In this case, the three lines of code that do the work of installing the agent server and publishing it are put inside the functor `RF`, and `RF` is installed remotely.

Now let's get access to both of these agent servers. We use the procedure `GetServer` that is defined in the functor `AgentServer`. This procedure can get access to *any* agent server, not just the one on its site:

```

declare
  Server1={GetServer "http://www.info.ucl.ac.be/~pvr/as1"}
  Server2={GetServer "http://www.info.ucl.ac.be/~pvr/as2"}

```

4.1.2 Programming agents

Now that the agent servers are installed, we're ready to let the agents work.

A first example creates an agent on the remote site. The agent queries the operating system and returns the value of the `OS.time` operation. Let's first execute the agent interactively:

```

declare D1 in
  {Server2 functor import OS define D1={OS.time} end}
  {Browse D1}

```

This first creates variable `D1` and then executes an agent on the remote site. The agent needs only one resource, `OS`, a module that gives access to some operating system functions. The agent is created asynchronously, which means that `D1` will usually still be unbound when the `Browse` is called. In most cases this is not a problem. The dataflow semantics of Oz mean that most operations that need `D1` will wait before continuing. If the caller wants to be sure that `D1` is bound before continuing, it just needs to do a `{Wait D1}`.

Now let's slightly modify the first example to get a standalone agent, i.e., the agent is itself a functor that can be compiled and executed standalone or by another application. We assume that the functor `AgentServer` is available at a standard place.

```

functor
import System AgentServer
define
  GetServer=AgentServer.getserver
  Server2={GetServer "http://www.info.ucl.ac.be/~pvr/as2"}

```

```

D1
{Server2 functor import OS define D1={OS.time} end}
{System.show D1}
end

```

This is almost the same as the previous example; it just extends it with a call to `GetServer` to access the agent server. The agent first gets access to the remote agent server `Server2` and then starts a calculation there.

If the functor `AgentServer` is not available in a standard place, then the standalone agent has to get it explicitly:

```

functor
import System Pickle Module
define
  AgentServer={Pickle.load "http://www.info.ucl.ac.be/~pvr/agents.ozf"}
  [AS]={Module.apply [AgentServer]}
  GetServer=AS.getserver
  Server2={GetServer "http://www.info.ucl.ac.be/~pvr/as2"}

  D1
  {Server2 functor import OS define D1={OS.time} end}
  {System.show D1}
end

```

4.1.3 There and back again

We give an example of an agent that goes to a remote site, does a calculation there, and comes back with the result. Surprise, surprise, the calculation takes almost no local CPU time.

4.1.4 Round and round it goes, where it stops nobody knows

We give an example of an agent that visits a set of sites repeatedly. It chooses dynamically the next site to visit. In fact, during its execution it can even get access to new sites that it has never heard of before and visit them.

4.1.5 Barrier synchronization

An agent can delegate work by creating tasks dynamically, and then wait until all tasks are done. This is easy by using logic variables to synchronize on the termination.

4.1.6 Definition of `AgentServer`

The basic tool giving agent functionality is the functor `AgentServer`, which is defined as follows. When installed, this functor creates an agent server, a procedure to publish the agent server, and a procedure to get access to any published agent server.

```

functor
import Module Connection Pickle
export
    server:AS
    publishserver:PublishServer
    getserver:GetServer
define
    S P={NewPort S}
    proc {InstallFunctors S}
        case S
        of F|S2 then
            try
                [_] = {Module.apply [F]}
            catch _ then skip end
            {InstallFunctors S2}
        else skip end
    end
    thread {InstallFunctors S} end
    proc {AS F} {Send P F} end
    T={Connection.offerUnlimited AS}

    % Make agent server available through file FN:
    % Note that the agent server is asynchronous.
    proc {PublishServer FN}
        {Pickle.save T FN}
    end

    % Get access to a server that is at file/URL UFN:
    proc {GetServer UFN AS}
        try
            T={Pickle.load UFN}
        in
            AS={Connection.take T}
        catch _ then
            raise serverUnavailable end
        end
    end
end

```

One way to use `AgentServer` is to compile it with the standalone compiler and make the resulting `ozf` file globally-accessible by putting it in a `public_html` directory. This can also be done in the interactive user interface:

```

declare
AgentServer=
    functor
        ... (body of functor definition)
    end

{Pickle.save AgentServer "/usr/staff/pvr/public_html/agents.ozf"}

```

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.

5.1 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`).

¹Chapter *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.

5.1.1 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.

5.1.2 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)`).

5.2 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 5.2.4).

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 5.4.1). 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 5.3.

5.2.1 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 5.2.4.

```
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.

5.2.2 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`.

²Since `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.

5.2.3 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 re-tried. 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.

5.2.4 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.

5.2.5 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.

³Since `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.

5.3 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*.

5.3.1 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 5.2.2. 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`.

⁴Note 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.

5.3.2 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.

5.4 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`.

5.4.1 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.

5.4.2 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`.

5.4.3 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.

Section 6.1.1 shows how to build a `SafeSend` abstraction that only completes on the sender site if it completes on the owner site.

5.4.4 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.

5.4.5 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(permSome)`).
- 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.

5.4.6 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`.

Fault-Tolerant Examples

This chapter shows how to use the failure model to build robust distributed applications. We first present basic fault-tolerant versions of common language operations. Then we present fault-tolerant versions of the server examples. We conclude with a bigger example: reliable objects with recovery.

6.1 A fault-tolerant hello server

Let's take a fresh look at the hello server. How can we make it resistant to distribution faults? First we specify the client and server behavior. The server should continue working even though there is a problem with a particular client. The client should be informed in finite time of a server problem by means of a new exception, `serverError`.

We show how to rewrite this example with the basic failure model. In this model, the system raises exceptions when it tries to do operations on entities that have problems related to distribution. All these exceptions are of the form `system(dp(conditions:FS ...) ...)` where `FS` is the list of actual fault states as defined before. By default, the system will raise exceptions only on the fault states `tempFail` and `permFail`.

Assume that we have two new abstractions:

- `{SafeSend Prt X}` sends to a port and raises the exception `serverError` if this is permanently impossible.
- `{SafeWait X T}` waits until `x` is instantiated and raises the exception `serverError` if this is permanently impossible or if the time `T` is exceeded.

We first show how to use these abstractions before defining them in the basic model. With these abstractions, we can write the client and the server almost exactly in the same way as in the non-fault-tolerant case. Let's first write the server:

```
declare Str Prt Srv in
{NewPort Str Prt}
thread
  {ForAll Str
   proc {$ S}
     try
       S="Hello world"
```

```

        catch system(dp(...) ...) then skip end
    end}
end

proc {Srv X}
    {SafeSend Prt X}
end

{Pickle.save {Connection.offerUnlimited Srv}
 "/usr/staff/pvr/public_html/hw"}

```

This server does one distributed operation, namely the binding `S="Hello world"`. We wrap this binding to catch any distributed exception that occurs. This allows the server to ignore clients with problems and to continue working.

Here's the client:

```

declare Srv

try X in
    try
        Srv={Connection.take {Pickle.load "http://www.info.ucl.ac.be/~pvr/hw"}}
    catch _ then raise serverError end
end

{Srv X}
{SafeWait X infinity}
{Browse X}
catch serverError then
    {Browse 'Server down'}
end

```

This client does two distributed operations, namely a send (inside `Srv`), which is replaced by `SafeSend`, and a wait, which is replaced by `SafeWait`. If there is a problem sending the message or receiving the reply, then the exception `serverError` is raised. This example also raises an exception if there is any problem during the startup phase, that is during `Connection.take` and `Pickle.load`.

6.1.1 Definition of `SafeSend` and `SafeWait`

We define `SafeSend` and `SafeWait` in the basic model. To make things easier to read, we use the two utility functions `FOneOf` and `FSomeOf`, which are defined just afterwards. `SafeSend` is defined as follows:

```

declare
proc {SafeSend Prt X}
    try
        {Send Prt X}
    catch system(dp(conditions:FS ...) ...) then

```

```

        if {FOneOf permFail FS} then
            raise serverError end
        elseif {FOneOf tempFail FS} then
            {Delay 100} {SafeSend Prt X}
        else skip end
    end
end

```

This raises a `serverError` if there is a permanent server failure and retries indefinitely each 100 ms if there is a temporary failure.

`SafeWait` is defined as follows:

```

declare
local
    proc {InnerSafeWait X Time}
        try
            cond {Wait X} then skip
            [] {Wait Time} then raise serverError end
        end
        catch system(dp(conditions:FS ...) ...) then
            if {FSomeOf [permFail remoteProblem(permSome)] FS} then
                raise serverError end
            if {FSomeOf [tempFail remoteProblem(tempSome)] FS} then
                {Delay 100} {InnerSafeWait X Time}
            else skip end
        end
    end
in
    proc {SafeWait X TimeOut}
        Time in
            if TimeOut\=infinity then
                thread {Delay TimeOut} Time=done end
            end
            {Fault.enable X 'thread'(this)
             [permFail remoteProblem(permSome) tempFail remoteProblem(tempSome)] _
             {InnerSafeWait X Time}
            }
        end
    end
end

```

This raises a `serverError` if there is a permanent server failure and retries each 100 ms if there is a temporary failure. The client and the server are the only two sites on which `x` exists. Therefore `remoteProblem(permFail:_ ...)` means that the server has crashed.

To keep the client from blocking indefinitely, it must time out. We need a time-out since otherwise a client will be stuck when the server drops it like a hot potato. The duration of the time-out is an argument to `SafeWait`.

6.1.2 Definition of `FOneOf` and `FSomeOf`

In the above example and later on in this chapter (e.g., in Section 6.2.3), we use the utility functions `FOneOf` and `FSomeOf` to simplify checking for fault states. We specify these functions as follows.

The call `{FOneOf permFail AFS}` is true if the fault state `permFail` occurs in the set of actual fault states `AFS`. Extra information in `AFS` is not taken into account in the membership check. The function `FOneOf` is defined as follows:

```
declare
fun {FOneOf F AFS}
  case AFS of nil then false
  [] AF2|AFS2 then
    case F#AF2
    of permFail#permFail(...) then true
    [] tempFail#tempFail(...) then true
    [] remoteProblem(I)#remoteProblem(I ...) then true
    else {FOneOf F AFS2}
    end
  end
end
```

The call `{FSomeOf [permFail remoteProblem(permSome)] AFS}` is true if either `permFail` or `remoteProblem(permSome)` (or both) occurs in the set `AFS`. Just like for `FOneOf`, extra information in `AFS` is not taken into account in the membership check. The function `FSomeOf` is defined as follows:

```
declare
fun {FSomeOf FS AFS}
  case FS of nil then false
  [] F2|FS2 then
    {FOneOf F2 AFS} orelse {FSomeOf FS2 AFS}
  end
end
```

6.2 Fault-tolerant stationary objects

To be useful in practice, stationary objects must have well-defined behavior when there are faults. We propose the following specification for the stationary object (the "server") and a caller (the "client"):

- The call `C={NewSafeStat Class Init}` creates a new server `C`.
- If there is no problem in the distributed execution then the call `{C Msg}` has identical language semantics to a centralized execution of the object, including raising the same exceptions.
- If there is a problem in the distributed execution preventing its successful completion, then the call `{C Msg}` will raise the exception `remoteObjectError`. It is unspecified how much of the object's method was executed before the failure.

- If there is a problem communicating with the client, then the server tries to communicate with the client during a short time period and then gives up. This does not affect the continued execution of the server.

We present two quite different ways of implementing this specification, one based on guards (Section 6.2.2) and the other based on exceptions (Section 6.2.3). The guard-based technique is the shortest and simplest to understand. The exception-based technique is similar to what one would do in standard languages such as Java.

But first let's see how easy it is to create and use a remote stationary object.

6.2.1 Using fault-tolerant stationary objects

We show how to use `Remote` and `NewSafeStat` to create a remote stationary object. First, we need a class—let's define a simple class `Counter` that implements a counter.

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

Then we define a functor that creates an instance of `Counter` with `NewSafeStat`. Note that the object is not created yet. It will be created later, when the functor is applied.

```
declare
F=functor
  import Fault
  export statObj:StatObj
  define
    {Fault.defaultEnable nil _}
    StatObj={NewSafeStat Counter init}
  end
```

Do not forget the "`import Fault`" clause! If it's left out, the system will try to use the local `Fault` on the remote site. This raises an exception since `Fault` is sited (technically, it is a *resource*, see Section 2.1.5). The `import Fault` clause ensures that installing the functor uses the `Fault` of the installation site.

It may seem overkill to use a functor just to create a single object. But the idea of functors goes much beyond this. With `import`, functors can specify which resources to use on the remote site. This makes functors a basic building block for mobile computations (and mobile agents).

Now let's create a remote site and make an instance of `Counter` called `StatObj`. The class `Remote.manager` gives several ways to create a remote site; this example uses the option `fork:sh`, which just creates another process on the same machine. The process is accessible through the module manager `MM`, which allows to install functors on the remote site (with the method "`apply`").

```

declare
MM={New Remote.manager init(fork:sh)}
StatObj={MM apply(F $)}.statObj

```

Finally, let's call the object. We've put the object calls inside a **try** just to demonstrate the fault-tolerance. The simplest way to see it work is to kill the remote process and to call the object again. It also works if the remote process is killed during an object call, of course.

```

try
  {StatObj inc}
  {StatObj inc}
  {Show {StatObj get($)}}
catch X then
  {Show X}
end

```

6.2.2 Guard-based fault tolerance

The simplest way to implement fault-tolerant stationary objects is to use a *guard*. A guard watches over a computation, and if there is a distribution fault, then it gracefully terminates the computation. To be precise, we introduce the procedure **Guard** with the following specification:

- **{Guard E FS S1 S2}** guards entity **E** for fault states **FS** during statement **S1**, replacing **S1** by **S2** if a fault is detected during **S1**. That is, it first executes **S1**. If there is no fault, then **S1** completes normally. If there is a fault on **E** in **FS**, then it interrupts **S1** as soon as a faulty operation is attempted on any entity. It then executes statement **S2**. **S1** must not raise any distribution exceptions. The application is responsible for cleaning up from the partial work done in **S1**. Guards are defined in Section 6.2.2.1.

With the procedure **Guard**, we define **NewSafeStat** as follows. Note that this definition is almost identical to the definition of **NewStat** in Section 3.2.3. The only difference is that all distributed operations are put in guards.

```

46a <Guard-based stationary object 46a>≡
proc {MakeStat PO ?StatP}
  S P={NewPort S}
  N={NewName}
in
  % Client interface to server:
  <Client side 47a>
  % Server implementation:
  <Server side 47b>
end

proc {NewSafeStat Class Init Object}
  Object={MakeStat {New Class Init}}
end

```

The client raises an exception if there is a problem with the server:

47a **<Client side 47a>**≡

```

proc {StatP M}
R in
  {Fault.enable R 'thread'(this) nil _}
  {Guard P [permFail]}
  proc {$}
    {Send P M#R}
    if R==N then skip else raise R end end
  end
  proc {$} raise remoteObjectError end end}
end

```

The server terminates the client request gracefully if there is a problem with a client:

47b **<Server side 47b>**≡

```

thread
  {ForAll S
    proc{$ M#R}
      thread RL in
        try {PO M} RL=N catch X then RL=X end
        {Guard R [permFail remoteProblem(permSome)]}
        proc {$} R=RL end
        proc {$} skip end}
      end
    end}
end

```

There is a minor point related to the default enabled exceptions. This example calls `Fault.enable` before `Guard` to guarantee that no exceptions are raised on `R`. This can be changed by using `Fault.defaultEnable` at startup time for each site.

6.2.2.1 Definition of `Guard`

Guards allow to replace a statement `S1` by another statement `S2` if there is a fault. See Section 6.2.2 for a precise specification. The procedure `{Guard E FS S1 S2}` first disables all exception raising on `E`. Then it executes `S1` with a local watcher `W` (see Section 6.2.2.2). If the watcher is invoked during `S1`, then `S1` is interrupted and the exception `N` is raised. This causes `S2` to be executed. The unforgeable and unique name `N` occurs nowhere else in the system.

```

declare
proc {Guard E FS S1 S2}
  N={NewName}
  T={Thread.this}
  proc {W E FS} {Thread.injectException T N} end
in
  {Fault.enable E 'thread'(T) nil _}
  try

```

```

        {LocalWatcher E FS W S1}
    catch X then
        if X==N then
            {S2}
        else
            raise X end
        end
    end
end
end

```

6.2.2.2 Definition of LocalWatcher

A *local watcher* is a watcher that is installed only during the execution of a statement. When the statement finishes or raises an exception, then the watcher is removed. The procedure `LocalWatcher` defines a local watcher according to the following specification:

- `{LocalWatcher E FS W S}` watches entity `E` for fault states `FS` with watcher `W` during the execution of `S`. That is, it installs the watcher, then executes `S`, and then removes the watcher when execution leaves `S`.

```

declare
proc {LocalWatcher E FS W S}
    {Fault.installWatcher E FS W _}
    try
        {S}
    finally
        {Fault.deInstallWatcher E W _}
    end
end
end

```

6.2.3 Exception-based fault tolerance

We show how to implement `NewSafeStat` by means of exceptions only, i.e., using the basic failure model. First `New` makes an instance of the object and then `MakeStat` makes it stationary. In `MakeStat`, we distinguish four parts. The first two implement the client interface to the server.

48a **⟨Exception-based stationary object 48a⟩**≡

```

declare
proc {MakeStat PO ?StatP}
    S P={NewPort S}
    N={NewName}
    EndLoop TryToBind
in
    % Client interface to server:
    ⟨Client call to the server 49a⟩
    ⟨Client synchronizes with the server 49b⟩
    % Server implementation:

```

```

    <Main server loop 50a>
    <Server synchronizes with the client 50b>
end

proc {NewSafeStat Class Init ?Object}
  Object={MakeStat {New Class Init}}
end

```

First the client sends its message to the server together with a synchronizing variable. This variable is used to signal to the client that the server has finished the object call. The variable passes an exception back to the client if there was one. If there is a permanent failure of the send, then raise `remoteObjectError`. If there is a temporary failure of the send, then wait 100 ms and try again.

49a <Client call to the server 49a>≡

```

proc {StatP M}
  R in
  try
    {Send P M#R}
  catch system(dp(conditions:FS ...) ...) then
    if {FOneOf permFail FS} then
      raise remoteObjectError end
    elseif {FOneOf tempFail FS} then
      {Delay 100}
      {StatP M}
    else skip end
  end
  {EndLoop R}
end

```

Then the client waits for the server to bind the synchronizing variable. If there is a permanent failure, then raise the exception. If there is a temporary failure, then wait 100 ms and try again.

49b <Client synchronizes with the server 49b>≡

```

proc {EndLoop R}
  {Fault.enable R 'thread'(this)
  [permFail remoteProblem(permSome) tempFail remoteProblem(tempSome)] _}
  try
    if R==N then skip else raise R end end
  catch system(dp(conditions:FS ...) ...) then
    if {FSomeOf [permFail remoteProblem(permSome)] FS} then
      raise remoteObjectError end
    elseif {FSomeOf [tempFail remoteProblem(tempSome)] FS} then
      {Delay 100} {EndLoop R}
    else skip end
  end
end

```

The following two parts implement the server. The server runs in its own thread and creates a new thread for each client call. The server is less tenacious on temporary failures than the client: it tries once every 2000 ms and gives up after 10 tries.

50a **⟨Main server loop 50a⟩**≡

```

thread
  {ForAll S
    proc {$ M#R}
      thread
        try
          {PO M}
          {TryToBind 10 R N}
        catch X then
          try
            {TryToBind 10 R X}
          catch Y then skip end
        end
      end
    end}
  end
end

```

50b **⟨Server synchronizes with the client 50b⟩**≡

```

proc {TryToBind Count R N}
  if Count==0 then skip
  else
    try
      R=N
      catch system(dp(conditions:FS ...) ...) then
        if {FOneOf tempFail FS} then
          {Delay 2000}
          {TryToBind Count-1 R N}
        else skip end
      end
    end
  end
end

```

6.3 A fault-tolerant broadcast channel

We can use the fault-tolerant stationary object (see Section 6.2) to define a simple open fault-tolerant broadcast channel. This is a useful abstraction; for example it can be used as the heart of a chat tool such as IRC. The service has a client/server structure and is aware of permanent crashes of clients or the server. In case of a client crash, the system continues to work. In case of a server crash, the service will no longer be available. Clients receive notification of this.

Users access the broadcast service through a local client. The user creates the client by using a procedure given by the server. The client is accessed as an object. It has a method `sendMessage` for broadcasting a message. When the client receives a message or is notified of a client or server crash, it informs the user by calling a user-defined procedure with one argument. The following events are possible:

- `permServer`: the broadcast channel server has crashed.
- `permClient(UserID)`: the client identified by `UserID` has crashed.
- `message(UserID Mess)`: receive the message `Mess` from client `UserID`.
- `registered(UserID)`: the client identified by `UserID` has registered to the channel.
- `unregistered(UserID)`: the client identified by `UserID` has unregistered from the channel.

We give an example of how the broadcast channel is used, and we follow this by showing its implementation. We first show how to use and implement a *non-fault-tolerant* broadcast channel, and then we show the small extensions needed for it to detect client and server crashes.

6.3.1 Sample use (no fault tolerance)

First we create the channel server. To connect with clients, the server offers a ticket with unlimited connection ability. The ticket is available through a publicly-accessible URL.

```
local
  S={NewStat ChannelServer init(S)}
in
  {Pickle.save {Connection.offerUnlimited S}
   "/usr/staff/pvr/public_html/chat"}
end
```

A client can be created on another site. We first define on the client's site a procedure `HandleIncomingMessage` that will handle incoming messages from the broadcast channel. Then we access to the channel by its URL. Finally, we create a local client and give it our handler procedure.

```
local
  proc {HandleIncomingMessage M}
    {Show {VirtualString.toString
          case M
            of message(From Content) then From# : '#Content
            [] registered(UserID)    then UserID# 'joined us'
            [] unregistered(UserID)  then UserID# 'left us'
          end}}
  end
  S={Connection.take {Pickle.load "http://www.info.ucl.ac.be/~pvr/chat"}}
  MakeNewClient={S getMakeNewClient($)}
  C={MakeNewClient HandleIncomingMessage 'myNameAsID'}
in
  {For 1 1000 1
   proc {$ I}
```

```

        {C sendMessage( 'hello' #I )}
        {Delay 800}
    end}

    {C close}
end

```

In this example we send 1000 messages of the form 'hello' #I, where I takes successive values from 1 to 1000. Then we close the client.

A nice property of this channel abstraction is that the client site only needs to know the channel's URL and its interface. All this can be stored in Ascii form and transmitted to the client at any time. In particular, the syntax of the interfaces, i.e., the messages understood by user, client, and server, is defined completely by a simple Ascii list of the message names and their number of arguments. The client site does not need to know any program code. When a client is created through a call to `MakeNewClient`, then at that time the client code is transferred from the channel server to the client site.

6.3.2 Definition (no fault tolerance)

Since a fault-tolerant stationary object has well-defined behavior in the case of a permanent crash, we can show the service's implementation in two steps. First, we show how it is written without taking fault tolerance into account. Second, we complete the example by adding fault handling code. This is easy; it amounts to catching the `remoteObjectError` exception for each remote method call (client to server and server to client).

The client and server are stationary objects with the following structure:

```

52a  <Client and server classes 52a>≡
      class ChannelClient
      feat
        server selfStatic usrMsgHandler userID
      <Client interface to user 53b>
      <Client interface to server 53a>
      end

      local
      <Concurrent ForAll procedure 54a>
      in
        class ChannelServer
          prop locking
          feat selfStatic
          attr clientList
          meth init(S)
            lock
              self.selfStatic=S
              clientList<-nil
            end
          end
        <Server's getMakeClient method 53c>
      end
    end

```

```

        <Server interface to client 54b>
    end
end

```

6.3.2.1 Client definition

The client provides two methods to the server. The first, `put`, for receiving broadcasted message from registered clients. The second, `init`, for the client initialization (remember that a client is created using a procedure defined by the server).

```

53a <Client interface to server 53a>≡
    meth put(Msg)
        {self.usrMsgHandler Msg}
    end

    meth init(Server SelfReference UsrMsgHandler UserID)
        self.server=Server
        self.selfStatic=SelfReference
        self.usrMsgHandler=UsrMsgHandler
        self.userID=UserID
        {self.server register(self.selfStatic self.userID)}
    end

```

The client keeps a reference to the server, to itself for unregistering, to the user-defined handler procedure, and to its user identification.

A user accesses the broadcast channel only through a client. The client provides the user with a method for sending a message through the channel and a method for leaving the channel.

```

53b <Client interface to user 53b>≡
    meth sendMessage(Msg)
        {self.server broadcast(self.userID Msg)}
    end

    meth close
        {self.server unregister(self.selfStatic self.userID)}
    end

```

6.3.2.2 Server definition

The server's `getMakeClient` method returns a reference to a procedure that creates clients:

```

53c <Server's getMakeClient method 53c>≡
    meth getMakeNewClient(MakeNewClient)
        proc {MakeNewClient UserMessageHandler UserID StaticClientObj}
            StaticClientObj={NewStat
                ChannelClient
                init(self.selfStatic StaticClientObj)
            }
        end
    end

```

```

                                UserID UserMessageHandler)}}
    end
end

```

The server uses a concurrent `ForAll` procedure that starts all sends concurrently and waits until they are all finished. This is important for implementing broadcasts. With the concurrent `ForAll`, the total time for the broadcast is the *maximum* of all client round-trip times, instead of the *sum*, if the broadcast would sequentially send to each client and wait for an acknowledgement before continuing. Concurrent broadcast is efficient in Mozart due to its extremely lightweight threads.

54a **⟨Concurrent ForAll procedure 54a⟩**≡

```

proc {ConcurrentForAll Ls P}
  Sync
  proc {LoopConcurrentForAll Ls PrevSync FinalSync}
    case Ls
    of L|Ls2 then
      NewSync in
        thread {P L} PrevSync=NewSync end
        {LoopConcurrentForAll Ls2 NewSync FinalSync}
      [] nil then
        PrevSync=FinalSync
      end
    end
  end
in
  {LoopConcurrentForAll Ls unit Sync}
  {Wait Sync}
end

```

The server provides three methods for the client, namely `register`, `unregister`, and `broadcast`. A client can register to the broadcast channel by calling the `register` method and unregister by calling the `unregister` method. Note that clients are identified uniquely by references to the client object `Client`, and not by the client's user ID `UserID`. This means that the channel will work correctly even if there are clients with the same user ID. The users may get confused, but the channel will not.

A client can broadcast a message on the channel by calling the `broadcast` method. The server will concurrently forward the message to all registered clients. The broadcast call will block until the message has reached all the clients.

54b **⟨Server interface to client 54b⟩**≡

```

meth register(Client UserID)
  CL in
    lock
      CL=@clientList
      clientList <- c(ref:Client id:UserID)|@clientList
    end
    {ConcurrentForAll CL
      proc {$ Element} {Element.ref put(registered(UserID))} end}
  end
end

```

```

meth unregister(Client UserID)
CL in
  lock
    clientList <-
      {List.filter @clientList
        fun {$ Element} Element.ref\=Client end}
    CL=@clientList
  end
  {ConcurrentForAll CL
    proc {$ Element} {Element.ref put(unregistered(UserID))} end}
end

meth broadcast(SenderID Msg)
  {ConcurrentForAll @clientList
    proc {$ Element} {Element.ref put(message(SenderID Msg))} end}
end

```

6.3.3 Sample use (with fault tolerance)

The fault-tolerant channel can be used in exactly the same way as the non-fault-tolerant version. The only difference is that the user-defined handler procedure can receive two extra messages, `permClient` and `permServer`, to indicate client and server crashes:

```

proc {UserMessageHandler Msg}
  {Show {VirtualString.toString
    case Msg
    of message(From Content) then From# : '#Content
    [] registered(UserID)      then UserID# ' joined us'
    [] unregistered(UserID)    then UserID# ' left us'
    [] permClient(UserID)      then UserID# ' has crashed'
    [] permServer              then 'Server has crashed'
    end}}
end

```

6.3.4 Definition (with fault tolerance)

The non-fault-tolerant version of Section 6.3.2 is easily extended to detect client and server crashes. First, the server and all clients must be created by calling `NewSafeStat` instead of `NewStat`. This means creating the server as follows:

```
S={NewSafeStat ChannelServer init(S)}
```

This makes the channel server a fault-tolerant stationary object. In addition, several small extensions to the client and server definitions are needed. This section gives these extensions.

6.3.4.1 Client definition

This definition extends the definition given in Section 6.3.2. We assume that the server has been created with `NewSafeStat`. Two changes are needed to the client. First, the client can detect a server crash by catching the `remoteObjectError` exception. Second, the server can detect a client crash in the same way, when it calls the client's `selfStatic` reference. Both of these changes can be done by redefining the values of `self.server` and `self.selfStatic` at the client.

```
meth init(Server SelfReference UsrMsgHandler UserID)
  self.server =
    proc {$ Msg}
      try
        {Server broadcast(self.userID Msg)}
      catch remoteObjectError then
        {self.usrMsgHandler permServ}
      end
    end
  self.selfStatic =
    proc {$ Msg}
      try
        {SelfReference Msg}
      catch remoteObjectError then
        {Server unregister(self.selfStatic self.userID)}
        {Server broadcastCrashEvent(UserID)}
      end
    end
  self.usrMsgHandler=UsrMsgHandler
  self.userID=UserID
  {self.server register(self.selfStatic self.userID)}
end
```

6.3.4.2 Server definition

The server has the new method `broadcastCrashEvent`.

```
meth broadcastCrashEvent(CrashID)
  {ConcurrentForAll @clientList
    proc {$ Element}
      {Element.ref put(permClient(CrashID))}
    end}
end
```

In the old method `getMakeNewClient`, the procedure `MakeNewClient` has to be changed to call `NewSafeStat` instead of `NewStat`:

```
meth getMakeNewClient(MakeNewClient)
  proc {MakeNewClient UserMessageHandler UserID StaticClientObj}
    StaticClientObj={NewSafeStat
```

```
ChannelClient
init(self.selfStatic StaticClientObj
      UserID UserMessageHandler)}
end
end
```

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.

7.1 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.

7.2 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.

7.3 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.

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

Bibliography

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

Index

- advanced failure model, 35
- agent
 - agent, agent server, 24
 - agent, barrier synchronization example, 23
 - agent, getting an agent server, 24
 - agent, publishing an agent server, 24
 - agent, resource, 23
 - agent, simple example, 25
 - agent, task, 23
- agent, 23
- agent server
 - agent server, definition, 26
- agent server, 24
- application
 - application, connected, 2
 - application, overall structure, 8
- array
 - array, sited entity, 7
- asynchronous failure detection, 36
- asynchronous many-to-one channel, 4
- atom, 6
- base module, 7
- basic failure model, 31
- C language
 - C language, relation to Oz cell, 5
- cached object, 3
- cell
 - cell, analogy to C and Java, 5
 - cell, fault states, 38
- centralized semantics, 3
- channel
 - channel, port (asynchronous many-to-one), 4
- chunk, 6
- class, 6
- communication failure, 29
- component-based programming, 9
- compute server, 16
- computed functor, 14
- concurrency
 - concurrency, 90% rule, 6
- connected applications, 2
- Connection module
 - `Connection` module, example, 12
- `Connection` module, 1
- constrained variable, 7
- CORBA, 1
- cross-site cycle, 8
- cycle, 8
- data-flow, 4
- default failure detection, 31
- dictionary
 - dictionary, sited entity, 7
- distributed semantics
 - distributed semantics, array, 7
 - distributed semantics, atom, 6
 - distributed semantics, chunk, 6
 - distributed semantics, class, 6
 - distributed semantics, definition, 3
 - distributed semantics, dictionary, 7
 - distributed semantics, function, 6
 - distributed semantics, functor, 6
 - distributed semantics, list, 6
 - distributed semantics, name, 6
 - distributed semantics, number, 6
 - distributed semantics, object, 7
 - distributed semantics, object-record, 7
 - distributed semantics, procedure, 6
 - distributed semantics, record, 6
 - distributed semantics, space, 7
 - distributed semantics, string, 6
 - distributed semantics, thread, 7
- eager
 - eager, failure detection, 36
 - eager, stateless data, 36
- example
 - example, client crash, 41
 - example, using `Connection` module, 12
 - example, using `Pickle` module, 12
 - example, using `Remote` module, 13

- exception
 - exception, format for failure exception, 33
 - exception, in failure model, 31
- fail-silent assumption, 29
- failure
 - failure, advanced model, 35
 - failure, basic model, 31
 - failure, binding logic variables, 32
 - detection
 - failure, detection, default, 31
 - failure, exception format, 33
 - failure, fail-silent, 29
 - failure, handler (synchronous detection), 35
 - failure, partial failure property, 29
 - failure, permanent network problem, 29
 - failure, temporary network problem, 29
 - failure, watcher (asynchronous detection), 36
- fault detection
 - fault detection, level, 34
- fault handler, 35
- fault level
 - fault level, relation with sitedness, 34
- fault level, 34
- `Fault` module, 2
- fault state
 - fault state, cell, 38
 - fault state, eager stateless data (records, procedures, functions, classes, functors), 36
 - fault state, exhaustive list, 31
 - fault state, lock, 38
 - fault state, logic variable, 37
 - fault state, object, 38
 - fault state, port, 37
 - fault state, sited entities, 37
- fault state, 29
- fault watcher, 36
- feeling
 - feeling, warm, fuzzy, 8
- finite domain variable, 7
- finite set variable, 7
- free record variable, 7
- function, 6
- functor
 - functor, computed, 14
- functor, 6
- future, 6, 7
- garbage collection
 - garbage collection, distributed, 8
 - garbage collection, in a server, 19
- geographic distribution, 3
- global name
 - global name, external, 11
 - global name, hostname, 11
 - global name, internal, 11
 - global name, ticket, 11
 - global name, URL, 11
- halt
 - halt, abnormal, 8
 - halt, normal, 8
- handler, 35
- home site, 7
- hostname
 - hostname, definition, 11
- Internet, 1
- Java language
 - Java language, limitation, 18
 - Java language, relation to Oz cell, 5
- Java language, 1
- language entity
 - language entity, module, 7
 - language entity, sited, 7
 - language entity, unsited, 7
- language semantics, 3
- latency tolerance, 5
- lazy
 - lazy, class, 4
 - lazy, failure detection, 35
- level
 - level, relation with sitedness, 34
- level of fault detection, 34
- lightweight thread, 4
- list, 6
- lock
 - lock, fault states, 38
 - lock, thread-reentrant, 5
- logic variable

- logic variable, binding in failure model, 32
 - logic variable, constrained, 7
 - logic variable, constrained variable, 7
 - logic variable, fault states, 37
 - logic variable, finite domain, 7
 - logic variable, free record, 7
 - logic variable, read-only (future), 6, 7
- logic variable, 5
- memory management, 8
- mobile agent, 23
- mobile object, 3
- module
 - base
 - module, base, `Cell`, 7
 - module, base, definition, 7
 - module, base, `Float`, 7
 - module, base, `Int`, 7
 - module, base, `List`, 7
 - module, base, `Lock`, 7
 - module, base, `Number`, 7
 - module, base, `Port`, 7
 - module, base, `Procedure`, 7
 - module, base, `Record`, 7
 - module, definition, 7
 - system
 - module, system, `Application`, 8
 - module, system, `Browser`, 8
 - module, system, `Connection`, 1, 8
 - module, system, definition, 7
 - module, system, `Fault`, 2
 - module, system, `FD`, 8
 - module, system, `Module`, 8
 - module, system, `Open`, 8
 - module, system, `OS`, 8
 - module, system, `Pickle`, 2, 8
 - module, system, `Property`, 8
 - module, system, `Remote`, 2, 8
 - module, system, `Search`, 8
 - module, system, `Tk`, 8
- multicast, 5
- multitasking, 3
- name, 6
- network failure, 29
- network-transparent, 3
- number, 6
- object
 - object, cached, 3
 - object, fault states, 38
 - object, mobile, 3
 - object, object-record, 7
 - object, sequential asynchronous stationary, 4
 - object, stationary, 4, 15
- owner site, 4, 7
- partial failure property, 29
- pickle, 11
- Pickle module
 - `Pickle` module, example, 12
- `Pickle` module, 2
- port
 - port, fault states, 37
 - port, Oz vs. Unix, 5
- port, 4
- procedure, 6
- process, 3
- programming
 - programming, component-based, 9
- proxy site, 4
- record
 - record, module, 7
 - record, object-record, 7
- record, 6
- remote computation
 - remote computation, example, 13
 - remote computation, of a statement (needing resources), 13
 - remote computation, of a statement (not needing resources), 17
- Remote module
 - `Remote` module, example, 13
- `Remote` module, 2
- resource, 7
- semantics
 - semantics, distributed, 3
 - semantics, language, 3
- server
 - server, compute server, 16
 - server, dynamically-extensible, 18
- server, 14

- shutdown
 - shutdown, abnormal, 8
 - shutdown, normal, 8
- site
 - site, definition, 3
 - site, multitasking, 3
 - site, owner, 4
 - site, proxy, 4
- sited entity
 - sited entity, array, 7
 - sited entity, definition, 7
 - sited entity, dictionary, 7
 - sited entity, resource, 7
 - sited entity, space, 7
 - sited entity, thread, 7
- sitedness
 - sitedness, relation with fault detection, 34
- space
 - space, sited entity, 7
- stateless entity
 - stateless entity, pickle, 11
- stationary object
 - stationary object, definition, 4, 15
 - stationary object, example, 15
- stream, 6
- string, 6
- synchronous failure detection, 35
- system module, 7
- task, 23
- thread
 - thread, data availability, 6
 - thread, lightweight, 4
 - thread, reentrant lock, 5
 - thread, sited entity, 7
- thread, 4
- thread-reentrant lock, 5
- ticket
 - ticket, definition, 11
 - ticket, example, 12
- time-out
 - time-out, difference with temporary fault, 30
- Unix port, 5
- unsited entity, 7
- URL
 - URL, definition, 11
- variable
 - variable, `final` in Java, 5
 - variable, logic variable, 5
- watcher, 36