

Open Programming in Mozart

Christian Schulte

Version 1.2.3
December 1, 2001



Abstract

Oz is a concurrent language providing for functional, object-oriented, and constraint programming. This document is intended as a tutorial on how to program open applications for Mozart.

Open applications need access to the rest of the computational world by exchanging data with it. Mozart provides classes to handle files, sockets, and processes.

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
1.1	Local Computation Spaces	1
1.2	Conventions	2
1.3	The Examples	2
2	Data Structures	3
2.1	Strings	3
2.2	Virtual Strings	4
3	Files	5
3.1	Basic Patterns of Use	5
3.1.1	Reading a File	5
3.1.2	Closing a File	6
3.1.3	Writing a File	7
3.1.4	Exceptions	7
3.2	Example: Expanding TAB Characters	8
4	Input and Output of Text	11
4.1	Expanding TAB Characters Revisited	11
5	Sockets	13
5.1	Sockets in a Nutshell	13
5.1.1	Domain	13
5.1.2	Type	13
5.1.3	Protocol	14
5.2	Stream Sockets	14
5.2.1	Initiating a Connection	14
5.2.2	Convenient Connection Establishment	16
5.2.3	Exchanging Data	16

5.2.4	Concurrency Issues	16
5.2.5	Disconnecting	17
5.2.6	Exceptions	17
5.3	Accepting Multiple Connections	17
5.4	Datagram Sockets	18
5.4.1	Initialization	18
5.4.2	Exchanging Data	19
5.4.3	Peer	19
5.5	Example: Are We Working Right Now?	20
6	Running Processes	21
6.1	Example: A Shell in Oz	21

Introduction

This document describes how to connect Mozart applications to the rest of the computational world. Sometimes this is paraphrased as interoperability.

Interoperability is supported in Mozart by the modules `Open` and `OS`. The module `Open` provides the following classes:

1. The class `Open.file` for reading and writing files.
2. The class `Open.socket` for sending and receiving data over the Internet.
3. The class `Open.pipe` to create operating system processes and to communicate with them.
4. The class `Open.text` for reading and writing texts character by character or line by line. This class can be combined with any of the classes from above.

Each of these classes is described by a chapter on its own (Chapter 3, Chapter 5, Chapter 6, and Chapter 4). The chapters explain the basic concepts to use the classes and contain a small example. Reference information to the classes can be found as usual in the document *Chapter Files, Sockets, and Pipes: Open, (System Modules)*.

The module `OS` provides procedures for random numbers, for manipulating files, directories, sockets, and the like. The module makes functionality found in the operating system available within Oz. Since Oz runs both on Unix based and Windows based platforms, the functionality provided is limited to what is defined by the POSIX.1 Standard [1]. Its documentation can be found in *Chapter Operating System Support: OS, (System Modules)*.

It is important to understand the data structures needed for reading, writing, and sending. For this reason Chapter 2 discusses these data structures in some detail.

1.1 Local Computation Spaces

There is very little to say about local computation spaces and input/output:

input and output do *not* work in local computation spaces!

1.2 Conventions

Throughout this document we refer to Unix manual pages for further information. For example, `open(2)` means that you should look up the manual page with title ‘open’ in Section 2. Type `man 2 open`¹ to your Unix shell to see the manual page. In case you are running a Windows based system, information can be found in a good Unix book or a book describing the POSIX.1 standard (e.g., [3]).

1.3 The Examples

This document contains quite a number of examples. It is recommended to read this document and try the examples while reading. All the examples collected into a single file can be found here².

¹On some systems you have to type `man -s 2 open` instead.

²OpenProgramming.oz

Data Structures

Data written and sent from Mozart are virtual strings, whereas data read and received are strings. In the following we review these data types and show some operations which may prove helpful in the context of Open Programming. Additional information on these data types can be found in Chapter *Text*, (*The Oz Base Environment*).

2.1 Strings

Strings are lists of characters, with characters being small integers between 0 and 255. For the coding of characters with integers we use the ISO 8859-1 standard [2].

There are special operations on strings, like testing whether a given value `x` is a string, by `{IsString X}`. For transforming strings into atoms, integers, or floats the procedures

- `String.toAtom`,
- `String.toInt`, and
- `String.toFloat`

are provided. For testing whether a given string can be transformed into an atom, integer, or float, the procedures

- `String.isAtom`,
- `String.isInt`, and
- `String.isFloat`

are provided.

In addition, all functionality provided by the module `List` is applicable. It is especially useful in combination with the functionality provided by the module `Char`.

Suppose that after reading or receiving a string `S="C@!3E4aN#61!"` all characters which are neither uppercase nor lowercase letters must be filtered out. This is achieved by using a combination of `Filter` and `Char.isAlpha`. Feeding:

3a `<{Filter S} 3a>≡`

```
{Browse {String.toAtom {Filter S Char.isAlpha}}}
```

will show the cleaned information in the browser window.

Instead of creating an atom from the cleaned string, the browser offers functionality to display strings directly in a convenient form. The functionality can be selected in the Representation, Type option, see also “*The Oz Browser*”. For Example,

4a `<{Browse S} 4a>≡`

```
{Browse {Filter S Char.isAlpha}}
```

displays in the browser window the character sequence "ClEaN".

Remember that character codes for letters are syntactically supported. For instance, the string

```
"hi there"
```

can be written equivalently as

```
[&h &i & &t &h &e &r &e]
```

2.2 Virtual Strings

The data to be sent or to be written consists of a sequence of integers, atoms, floats, and strings. If only strings were allowed to be sent as information, the data would have to be transformed into a string. This would be clumsy and inefficient with regard to both space and time.

Mozart uses virtual strings for this purpose instead. A virtual string is either an atom, a float, an integer, a string, or a tuple with label ‘#’, whose subtrees are virtual strings themselves. For instance,

```
12#(2#fast#4#"U")#~3.1415
```

is a virtual string. In the above example it is quite clear, that # stands for concatenation (i.e., for *virtual* concatenation).

There is predefined functionality for virtual strings, like testing, by `IsVirtualString`, converting to a string, by `VirtualString.toString`, and changing of signs, by `VirtualString.changeSign`. The latter procedure is quite important, because

```
{Browse 12#(2#fast#4#"U")#~3.1415}
```

reveals that the usual unary minus sign (i.e., -) is used rather than the Oz operator ~.

The procedure `VirtualString.changeSign` provides the possibility to choose any virtual string as minus sign in numbers. It takes as input arguments two virtual strings, and substitutes every occurrence of the - character as a unary minus sign in numbers of the first argument by the second argument. For example,

```
{VirtualString.changeSign 12#(2#fast#4#"U")#~3.1415 '~'}
```

returns the virtual string with the Oz-style unary minus sign.

Note that in order to display virtual strings in readable form in the Oz Browser, you have to configure the Representation, Type option, see also “*The Oz Browser*”. In the following we assume that the Browser is configured for displaying virtual strings.

Files

This section explains by means of examples how to read data from and write data to a file.

3.1 Basic Patterns of Use

The basic idea of how to use files in Mozart is as follows. We provide a class `Open.file`. To an object created from this class we refer to as file object. Conceptually, its state consists of a string together with the current position in this string called seek pointer.

In the following, we will refer to this particular string as file. It is visible to other operating system processes via the operating system's filesystem.

The atoms used for labels and fields of methods of the class `Open.file` are chosen to coincide with the common operating system terminology (i.e., POSIX terminology).

3.1.1 Reading a File

Suppose we want to read data from a file named `a.txt`. The contents of this file is as follows¹ :

	0	1	2	3	4	5	6	7	8	9
0	H	e	l	l	o	H	e	l	l	o
10	O	z	_	I	s	_	b	e	a	u
20	t	i	f	u	l					

The first step is to create a file object and associate it to the file `a.txt`. This is done by feeding

```
F={New Open.file init(name:'a.txt' flags:[read])}
```

The list `[read]` used as value of the field `flags` means that we want to have read access to the file.

After the file has been opened and associated to the file object `F`, we can read from it. To read the next five characters from the file we feed

¹The demo file² for this document contains the Oz code to create this file.

²`OpenProgramming.oz`

```
{F read(list:{Browse} size:5)}
```

The character string `Hello` appears in the browser window (the value `5` at the field `size` signals that we want to read five characters from the file).

Note that if we had not switched the browser to the mode for displaying virtual strings (see Chapter 2), we would have seen the output `[72 101 108 108 111]` in the browser window.

A common pattern of use is to read the entire file into an Oz string. This is especially supported. Feed:

```
{F read(list:{Browse} size:all)}
```

The rest of the file `HelloOz is beautiful` appears in the browser window.

Up to now we have read the data in strict left to right fashion. Suppose we want to start reading at the eighth character of the file. Navigating to this character and reading the following five characters is done by:

```
{F seek(whence:set offset:7)}
{F read(list:{Browse} size:5)}
```

You will see `lloOz` in the browser window. Internally, as already mentioned, each file object has a seek pointer. Each operation refers to its position: for instance, instead of ‘reading five characters’ one should think of ‘reading the next five characters after the seek pointer’. Note that in order to read the first character of a file, the seek pointer must be set to the offset zero.

Furthermore, you can get the current position of the seek pointer by:

```
{F tell(offset:{Browse})}
```

The number `12` appears in the browser window.

3.1.2 Closing a File

After use, the file object `F` must be closed. By applying `F` to the message `close`:

```
{F close}
```

the file as well as the file object are closed. Note that invoking a method other than `close` of the class `Open.file` after the object has been closed raises an exception. Exceptions are discussed later.

3.1.3 Writing a File

Suppose we want to create an object for a file which does not yet exist and that should be named `ours.txt`. Furthermore, this file should have read and write access rights for yourself and for each member of your group. We feed

```
F={New Open.file
    init(name: 'ours.txt'
        flags: [write create]
        mode: mode(owner: [read write]
                    group: [read write]))}
```

Data to be written to the file must be virtual strings. For example, the character sequence `"Go ahead"` can be written to the file `F` by

```
{F write(vs:'Go ahead!')}
```

The same character sequence we also can write by

```
{F write(vs:"Go ahead!")}
```

Also more complex virtual strings are handled this way. For example

```
{F write(vs:"This is "#1#' And '#
            ("now a float: "#2.0)##"\n")})}
```

writes a nested virtual string containing integers, atoms, strings and floats to the file. Even the filename argument of the `init` method of class `Open.file` is allowed to be a virtual string. For more information on virtual strings see Section 2.2.

If you type the Unix command

```
cat ours.txt
```

or the Windows command

```
type ours.txt
```

to a shell, you see the content of `ours.txt` printed to standard output.

3.1.4 Exceptions

Functionality provided by the class `Open.file` relies on operating system services. These services might report exceptions, these exceptions are then raised as Oz exceptions.

For example, trying to open a non existing file raises an exception. Operating system dependent exceptions can be caught as follows:

```

try
  _={New Open.file init(name:'not-existing')}
catch system(os(A W I S) ...) then
  {Browse os(category:    A
                 what:    W
                 number:   I
                 description: S)}
end

```

where *A* is an atom describing the category of the error (e.g., *os* or *host*), *W* the system call that raised the exception (as string), *I* is the operating system dependent error number, and *S* is a string describing the error.

Besides of operating system exceptions, the methods of the class *Open.file* can raise two different exceptions themselves:

1. An exception is raised when an object is initialized twice. This exception can be caught as follows:

```

try
  ... % Initialize twice
catch system(open(alreadyInitialized O M) ...) then
  ...
end

```

Here *O* is the object that has been tried to be initialized twice by applying it to the method *M*.

2. An exception is raised when a message other than *close* is executed by an already closed file object. This exception can be caught as follows:

```

try
  ... % Apply closed object
catch system(open(alreadyClosed O M) ...) then
  ...
end

```

Here *O* is the object that has been closed already and *M* the message *O* has been applied to.

3.2 Example: Expanding TAB Characters

Suppose we want to read a file, expand all TAB characters to space characters, and write the expanded lines to another file. The program which implements this task is shown in Figure 3.1. The file with name *IN* is opened for reading. After reading the entire file into the list *Is*, the file and the associated object are closed. Remember that reading the entire file is obtained by giving *all* as the value for feature *size*.

The expansion of TAB characters is done in the function *Scan*. It takes as input parameters the list of characters *Is*, the *Tab*, and the current position *N* in the current line.

Figure 3.1: The `Expand` procedure.

```

local
  fun {Insert N Is}
    if N>0 then {Insert N-1 & |Is} else Is end
  end
  fun {Scan Is Tab N}
    case Is of nil then nil
    [] I|Ir then
      case I
      of &\t then M=Tab-(N mod Tab) in {Insert M {Scan Ir Tab M+N}}
      [] &\n then I|{Scan Ir Tab 0}
      [] &\b then I|{Scan Ir Tab {Max 0 N-1}}
      else I|{Scan Ir Tab N+1}
      end
    end
  end
end
in
  proc {Expand Tab IN ON}
    IF={New Open.file init(name:IN)}
    OF={New Open.file init(name:ON flags:[write create truncate])}
    Is
  in
    {IF read(list:Is size:all)} {IF close}
    {OF write(vs:{Scan Is Tab 0})} {OF close}
  end
end

```

The outer `case` of `Scan` figures out whether there are characters to process. If the next character to process is a TAB character, enough space characters to reach the next multiple of `TabStop` are inserted. This is performed by the self explanatory function `Insert`.

A newline character resets the position `N` to zero. The position is decremented whenever a backspace character is encountered. Any other character increments the position.

A second file is opened for writing (indicated by `write`). If a file with name `ON` does not exist, it is created (indicated by `create`). Otherwise the already existing file is truncated to length zero (indicated by `truncate`) and rewritten. The expanded string is written to this file.

The file and its associated file object are closed after writing the expanded list of characters to it.

Input and Output of Text

The example in Section 3.2 resulted in a very inefficient program: reading the entire file to a list and processing it afterwards is extremely memory consuming.

A far better solution is to process the file incrementally line by line. This pattern of reading files occurs in fact very often, thus we provide support for it. The support provided is a mixin class `Open.text`. This mixin class can be used together with files, sockets, and pipes.

4.1 Expanding TAB Characters Revisited

In Figure 4.1 the revised formulation of the `Expand` procedure is shown. As before the file objects are created, but now both files inherit from `Open.text` as well. This class provides methods for buffered input and output.

The procedure `Scan` applies the input file object `IF` to the message `getS($)`. It yields either `false`, in case the end of the file is reached, or a string. This string contains exactly one line of the input file (without a newline character).

The expansion of TAB characters is done in the function `ScanLine` as before. The expanded lines are written with the `putS` method.

Figure 4.1: The Incremental `Expand` Procedure

```

local
  fun {Insert N Is}
    if N>0 then {Insert N-1 & |Is} else Is end
  end
  fun {ScanLine Is Tab N}
    case Is of nil then nil
    [] I|Ir then
      case I
      of &\t then M=Tab-(N mod Tab) in {Insert M {ScanLine Ir Tab M+N}}
      [] &\b then I|{ScanLine Ir Tab {Max 0 N-1}}
      else I|{ScanLine Ir Tab N+1}
      end
    end
  end
end
proc {Scan Tab IF OF}
  Is={IF getS($)}
in
  if Is==false then
    {IF close} {OF close}
  else
    {OF putS({ScanLine Is Tab 0})}
    {Scan Tab IF OF}
  end
end
class TextFile
  from Open.file Open.text
end
in
  proc {Expand Tab IN ON}
    {Scan Tab
      {New TextFile init(name:IN)}
      {New TextFile init(name: ON
                                flags: [write create truncate])}}
  end
end

```

Sockets

A frequent need in programming open applications is to connect different operating system processes such that they are able to exchange data. A tool for solving this problem are sockets. This chapter explains how to use sockets from Mozart.

Before starting to describe how to use them, we explain some basic concepts. The explanation is short; for more detailed information see for example [4].

5.1 Sockets in a Nutshell

A socket is a data structure which can be used for communication between operating system processes on possibly different computers connected by a network.

5.1.1 Domain

Unix based operating systems offers two domains for sockets: The Internet domain and the Unix domain. The first can be used for communication all over the world, whereas the latter is only applicable within one Unix system. Other systems might offer only sockets for the Internet domain. Mozart supports sockets for the Internet domain.

5.1.2 Type

Sockets we are going to describe here are either of type stream or of type datagram . There are more than these types of sockets, which are used by the operating system internally for implementing stream and datagram sockets themselves. See for instance `socket(2)`.

A stream socket is connection-oriented: Two processes establish a one-to-one connection in using a socket. After the connection has been established a stream socket has the following features:

1. Stream oriented: Character codes are incrementally sent and received.
2. Duplex: Both processes have sending and receiving access to the stream. The data passed are bytes, i.e. small integers between 0 and 255.
3. Reliability: No data will be lost in most cases.
4. Sequencing: The order of data sent will not be changed.

A datagram has only the feature of being duplex: It provides no reliability and no sequencing (messages may arrive in any order). It supports sending and receiving data packets of a (usually) small size.

5.1.3 Protocol

The protocol of a socket gives services used for performing communication on it. For example, the usual protocols for the Internet domain are TCP (Internet Transmission Control Protocol) for stream sockets and UDP (Internet User Datagram Protocol) for datagram sockets.

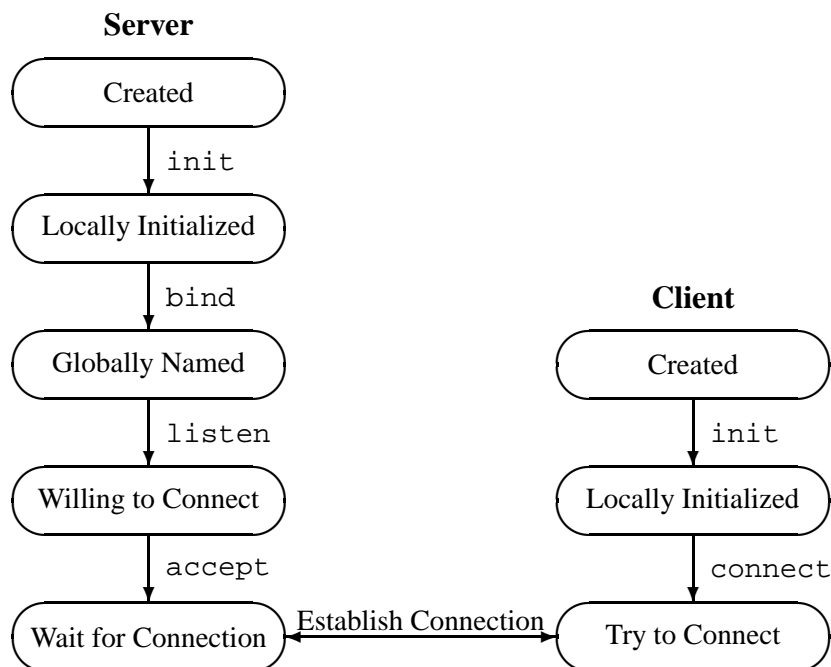
5.2 Stream Sockets

Instead of giving detailed explanations, we will follow an example tailored to understand how stream sockets for the Internet. For this purpose, we use the class `Open.socket`.

5.2.1 Initiating a Connection

Before any data between the server and the client can be exchanged, they must initiate a connection. How to obtain a connection is shown in Figure 5.1. We will perform the different steps shown in the figure as an example.

Figure 5.1: Initiating a Stream Connection.



The starting point for server and client are the topmost ovals on the two sides of the figure. We first turn our attention to the `Server` object. Basic initialization is performed by:

```
Server = {New Open.socket init}
```

After execution of this message, `Server` has initialized the local data structures it needs.

After locally initializing the socket we have to provide it with a global name. We use this global name for connecting our client to this server. Feed:

```
{Server bind(port:{Browse})}
```

The Internet protocol services generate a so-called port number. The port number is shown in the browser window. We refer to this number as `OurPort` later on.

The combination of the host computer on which Oz is running and the generated port number gives a unique address within the Internet domain. So any other party in the Internet domain may use this pair to refer unambiguously to this socket.

As next step, we signal that our socket is willing to accept connections from another socket. To do so, we feed:

```
{Server listen}
```

Now we are ready to accept our connection. Typing:

```
{Browse H#": "#P}  
{Server accept(host:H port:P)}
```

does nothing at the moment. But if a connection request is signaled at port `OurPort` the connection is accepted. In this case `H` is bound to a string giving the name of the computer the connection request was received from. In the same way `P` is bound to the respective port.

The next step connects the `Client` to the `Server`.

```
Client = {New Open.socket init}
```

initializes our client object.

By typing

```
{Client connect(host:localhost port:OurPort)}
```

our client connects to the port `OurPort` on the same computer (thus the virtual string `localhost`).

Since our server suspended on a connection request at port `OurPort` it now resumes, and the name of the host and the port appears in the Browser window.

5.2.2 Convenient Connection Establishment

This was the whole story in detail, but for convenience we provide two methods, which perform the complete connection establishment.

Instead of the various method applications, the following suffices for the server:

```
{Server server(host:H port:P)}
```

Here `P` is bound to the automatically generated port number. After a connection has been accepted on this port, `H` is bound to the string containing the name of the connecting host.

Symmetrically, for the client side, one simply can use:

```
{Client client(host: HostToConnectTo
               port: PortToConnectTo)}
```

5.2.3 Exchanging Data

Now we can await data at our server by feeding

```
{Server read(list:{Browse})}
```

Sending data from our client can be accomplished via

```
{Client write(vs:'Hello, good'#" old Server!")}
```

and in the browser window (on the server side) the message appears.

By simply flipping `Server` and `Client` in the last two expressions fed, you can send data from the server to the client.

Notice, however, that in this example server and client are created in the same operating system process on the same host computer. Instead we could have used two different processes and two different host computers even running two different operating systems.

5.2.4 Concurrency Issues

Both reading and writing to a socket might not be possible immediately. Possible reasons include:

1. The data to be written is not yet bound to a virtual string.
2. There is no data to be read at the moment.
3. The socket can not transfer data over the network at the moment.

In this case access to the socket has to be synchronized in some way. The following invariant is kept: all `read` messages are performed in the order they were received by the object. The same holds for all `write` messages. But there is no order between `reads` or `writes`. Both the `read` and the `write` method reduce only when the request can be served.

If you want to make sure that both all `read` and `write` requests are finished, you can use the method `flush`.

5.2.5 Disconnecting

If there is no further data to be exchanged between client and server, they should be disconnected. The simplest way to do so is by feeding

```
{Server close}
{Client close}
```

Suppose that the server closes before the client, and the client is still sending data to the server. In this case the data will still arrive at the socket. And only after some time the data will be thrown away, consuming valuable memory resources. To prevent from this, you can signal that you are not interested in receiving any messages, simply by typing (before closing the object!)

```
{Server shutDown(how: [receive])}
```

If we are not interested in sending data anymore, we can inform the operating system that we are indeed not willing to send anything more, thus

```
{Server shutDown(how: [send])}
```

Both applications could have been combined into

```
{Server shutDown(how: [receive send])}
```

5.2.6 Exceptions

The methods of the class `Open.socket` might raise exceptions of the same format as already described for the class `Open.file`, see Section 3.1.

5.3 Accepting Multiple Connections

A frequently occurring situation is to have a single server with a port number known to several clients and that all these clients want to connect to that server. For this purpose sockets have the possibility to accept more than a single connection.

The first step is to create the server socket `S` with a port number `P`:

```
S={New Open.socket init}
P={S bind(port:$)}
{S listen}
```

The clients can connect later to this socket with the port number `P`.

How to set up the server is shown in Figure 5.2. The procedure `Accept` waits until the server socket `S` accepts a connection. When a connection is accepted, the variable `A` is bound to a newly created object. The object `A` is created from the class `Accepted`, this is specified by the feature `acceptClass`. The newly created object is already connected: Applying the object to the message `report(H P)` waits that on the accepted connection data arrives.

The loop to accept connections is started by applying the procedure `Accept`:

Figure 5.2: Accepting multiple connections.

```

class Accepted from Open.socket
  meth report(H P)
    {Browse read(host:H port:P read:{self read(list:$)})}
    {self report(H P)}
  end
end

proc {Accept}
  H P A
in
  {S accept(acceptClass:Accepted
            host:?H port:?P accepted:?A)}
  thread
    {A report(H P)}
  end
  {Accept}
end

```

```
{Accept}
```

Let us assume that we have two clients `C1` and `C2` that need to connect to the socket with port number `P`:

```

C1={New Open.socket client(port:P)}
C2={New Open.socket client(port:P)}

```

After the clients are connected, sending data from the clients is as usual:

```
{C1 write(vs:'hello from C1')}
```

As soon as the socket object created by the procedure `Accept` receives the data sent from a client, the data appears in the Browser.

5.4 Datagram Sockets

Datagram sockets are for connectionless use. This implies that there is no distinction between server and client. The basic patterns of use are as follows.

5.4.1 Initialization

First we create and initialize two socket objects `S` and `T` with the type `datagram`:

```

S={New Open.socket init(type:datagram)}
T={New Open.socket init(type:datagram)}

```

Both sockets can be named in the Internet domain as follows:

```
SPN={S bind(port:$)}
TPN={T bind(port:$)}
```

5.4.2 Exchanging Data

To send the virtual string `'really '# "great"` from socket `S` to socket `T`, we feed

```
{S send(vs:'really '# "great" port:TPN)}
```

Data can be received as follows:

```
{T receive(list:?Is port:?P)}
{Browse Is# ' from: '#P}
```

and `Is` is bound to the string `"really great"` and `P` to the port number `SPN`. The roles of sender and receiver may be toggled as you like.

To send to a socket on a different computer, the hostname of the computer can be specified. For example,

```
{S send(vs:'donald' host:'duck.somewhere' port:4711)}
```

would send `'donald'` to the socket with port number `4711` on the computer `duck.somewhere`. Similarly, by using the field `host` for the `receive` method the sending computer can be found out.

5.4.3 Peer

An additional feature of the datagram socket is the ability to declare a socket as peer . Suppose we want to declare socket `S` as a peer of socket `T`. This is done by

```
{S connect(port:TPN)}
```

Now any message sent from socket `S` is received by socket `T`. Furthermore, it suffices to specify just the data to be sent. Hence

```
{S send(vs:"really great")}
```

has the same effect as the previous example. As an additional effect, only data sent by socket `T` is received at socket `S`. After a peer has been assigned, you can even use the methods `read` and `write`.

Disconnecting is the same as for stream sockets.

5.5 Example: Are We Working Right Now?

As an example, we show how to use the `finger` service from Oz. Services are programs running on a host, where communication is provided by a socket connection. A well known example is the `finger` service.

Services use the Internet domain, thus for connecting to them we need the name of the host computer (i.e. its Internet address) and the port of the socket. For this reason there is the procedure `OS.getServByName` (see Chapter *Operating System Support: OS, (System Modules)*) which gives the port of the socket on which the service is available. Feeding

```
FingerPort={OS.getServByName finger tcp}
```

binds `FingerPort` to the port number of the service.

Note that one has to specify the protocol the service uses, here the TCP protocol (as specified by the virtual string `tcp`). The port number of a service is unique for all computers, so this number does not depend on your localhost.

To get the information from this service we have to connect to the `finger` service. If you want to know whether the author is working right now, then you can connect to one of our computers with the Internet address `wallaby.ps.uni-sb.de` on the `FingerPort`. This is done by feeding the code shown in Program Figure 5.3.

Figure 5.3: The finger client `FC`.

```
class FingerClient from Open.socket
  meth getInfo($)
    Is Ir
  in
    if {self read(list:Is tail:Ir len:$)}==0 then nil
    else {self getInfo(Ir)} Is
    end
  end
end
FC = {New FingerClient client(host:'wallaby.ps.uni-sb.de'
                             port:FingerPort)}
```

For discovering whether the author is logged in, you have to send his username (in this example `schulte`) followed by a newline to the service, and then receive the information:

```
{FC write(vs:'schulte\n')}
{Browse {FC getInfo($)}}
```

For receiving the information we use the method `getInfo` which reads the entire information from the socket until it is closed (the `finger` service closes the connection automatically after sending its information).

Running Processes

A frequent need in programming open applications is to start an operating system process from Oz and to send information to and receive information from the running process.

For this purpose we provide a class `Open.pipe`.

6.1 Example: A Shell in Oz

Suppose, we want to start and control a Unix Bourne shell `sh` (see `sh(n)`) by an Oz program. We first have to start a Unix process running the shell and then we need a connection to this shell to send commands to it and receive its output.

This can be done with class `Open.pipe`. Program Figure 6.1 shows the definition of a shell class. Creating a shell object by

Figure 6.1: A shell class.

```
class Shell from Open.pipe Open.text
  meth init
    Open.pipe,init(cmd:"sh" args:["-s"])
  end
  meth cmd(Cmd)
    Open.text,putS(Cmd)
  end
  meth show
    case Open.text,getS($) of false then
      {Browse 'Shell has died.'} {self close}
    elseif S then {Browse S}
    end
  end
end
```

`S={New Shell init}`

the command `sh` is executed in a newly created process. The command `sh` gets the argument `-s` to signal that the shell should read from standard input. The forked process is connected by its standard input and standard output to the created `Shell` object.

By inheriting from the class `Open.text` we can simply send text to the running `sh` process by using the `puts` method, and receive lines of text by using the `gets` method.

If we want to see the files in our home directory, we will first navigate to it by `cd(1)`, and then issue the `ls(1)` command:

```
{S cmd(cd)}  
{S cmd(ls)}
```

Now we can incrementally request the names of the files by

```
{S show}
```

and they will appear in the Browser window.

Closing the object and the shell is done simply by

```
{S close}
```

Bibliography

- [1] IEEE. *Portable Operating System Interface for Computer Environments (POSIX)*. 1990. IEEE Standard 1003.1-1990.
- [2] Information processing – 8-bit single-byte coded graphic character sets – part 1: Latin, alphabet no. 1. Technical Report ISO 8859-1:1987, Technical committee: JTC 1/SC 2, International Organization for Standardization, 1987.
- [3] Donald Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates, Inc., April 1991.
- [4] W. Richard Stevens. *Unix Network Programming*. Software Series. Prentice Hall, 1990.

Index

- cat, 7
- Char
 - Char, isAlpha, 3
- Char, 3
- domain
 - domain, Internet, 13
- domain, 13
- domain
 - domain , Unix, 13
- Expand, 11
- file
 - file, object, 5
- Filter, 3
- finger, 20
- FingerClient, 20
- interoperability, 1
- IsString, 3
- IsVirtualString, 4
- List, 3
- localhost, 15
- manualpages, 2
- Open
 - file
 - Open, file, close, 6
 - Open, file, init, 5, 7
 - Open, file, read, 6
 - Open, file, seek, 6
 - Open, file, tell, 6
 - Open, file, write, 7
 - Open, pipe, 21
 - socket
 - Open, socket, accept, 15
 - Open, socket, bind, 15, 19
 - Open, socket, client, 16
 - Open, socket, close, 17
 - Open, socket, connect, 15, 19
 - Open, socket, init, 15, 19
 - Open, socket, listen, 15
 - Open, socket, read, 16
 - Open, socket, receive, 19
 - Open, socket, send, 19
 - Open, socket, server, 16
 - Open, socket, shutDown, 17
 - Open, socket, write, 16
- Open, socket, 15
- Open, text, 22
- OS
 - OS, getServByName, 20
- peer
 - peer, of a socket, 19
- protocol, 14, 20
- protocol
 - protocol , TCP, 14, 20
 - protocol , UDP, 14
- service, 20
- Shell, 21
- String
 - String, isAtom, 3
 - String, isFloat, 3
 - String, isInt, 3
 - String, toAtom, 3
 - String, toFloat, 3
 - String, toInt, 3
- string, 3
- strings
 - strings, virtual, 3, 4
- type (of a socket)
 - type (of a socket), datagram, 13
 - type (of a socket), stream, 13
- type (of a socket), 13
- VirtualString
 - VirtualString, changeSign, 4
 - VirtualString, toString, 4