CLIENT-SERVER DEVELOPMENT WITH EIFFEL:

THE EIFFELNET LIBRARY

ISE Technical Report Series

Report identification

ISE Technical Report TR-EI-47/EN: *Client-Server development with Eiffel: The EiffelNet library.*

Publication history

First published: August 1994.

Version: 3.2.4, August 1994. Corresponds to version 3.2.4 of the ISE Eiffel 3 environment.

Authors

Patrice Rolland and Bertrand Meyer.

Software credits

EiffelNet was developed by Glen Smith, David Morgan and Patrice Rolland.

Copyright notice and proprietary information

The software described in this report is available as part of the ISE Eiffel 3 environment, an advanced development system for the production of quality object-oriented software. ISE Eiffel 3 is available on many different hardware and software platforms. For more details about ISE Eiffel 3 and other ISE products, to obtain the name of a distributor in your area, or to receive information about Eiffel and object-oriented training, please contact:

Interactive Software Engineering Inc.
270 Storke Road Suite 7
Goleta, CA 93117
USA
Telephone 805-685-1006, Fax 805-685-6869
E-mail <info@eiffel.com>

CONTENTS

Client-Server Development with Eiffel:

The EiffelNet Library

1 INTRODUCTION TO EIFFELNET

1.1 What is EiffelNet?

The client-server model of computing is quickly gaining ground as a way of optimizing network resources and of building sophisticated and flexible transactional applications.

The object-oriented method of software development, as embodied in Eiffel, potentially provides ideal support for developing client-server applications. The EiffelNet library described in this report makes this potential a reality by addressing one the crucial needs of client-server development: how to exchange objects between software systems. Systems using EiffelNet can exchange object structures with other systems running on the same machine or on different machines; in the case of different machines, the hardware architectures need not be the same. Objects are transmitted using sockets.

Writing client-server applications, especially their communication aspects, is an arduous task; developers must master many tricky details of socket programming and other low-level mechanisms. EiffelNet hides this complexity by providing ready-made classes addressing the most common cases, so that developers only have to describe what is specific to their application, letting EiffelNet handle the whole process of communication and synchronization between clients and servers. This part of EiffelNet is known as the **predefined level** and for many uses this is the only one that you will need to learn about. For developments requiring a finer degree of control, all the major mechanisms of socket programming are still accessible in a convenient form through more specialized classes.

In offering these varying levels of abstraction and control, EiffelNet follows the principles set by other libraries of the ISE Eiffel 3 environment (see for example the discussion of EiffelLex in reference [2]). Some users require a mechanism that will be very easy to learn and will hide all the details in common cases; other users (or the same at later stages of their work) need to access the whole array of facilities available at the level of the underlying hardware or software. The various components of the library should address all these needs.

This manual describes how to use EiffelNet at its various levels of abstraction.

1.2 Background and prerequisites

EiffelNet is a component of the ISE Eiffel 3 development environment. Please contact your ISE Eiffel distributor or ISE to find out how to acquire EiffelNet if it is not already part of your delivery.

This manual assumes the following prerequisites:

- Familiarity with the basic Eiffel concepts: classes, inheritance, features, assertions, exception handling (see reference [1]).
- Experience of writing software with Eiffel.
- Basic knowledge of the EiffelBase libraries, as described in reference [2].

Familiarity with the basic concepts of client-server computing will also be helpful (see for example reference [3]; if you are new to this area, however, you will still be able to use EiffelNet, as this manual defines all the needed concepts.

1.3 Organization of this manual.

Section 2 discusses the notion of client and server. Section 3 presents an overview of EiffelNet's facilities. Section 4 describes the predefined level: a set of high-level classes that provide a complete framework, covering the most common uses and limiting the developer's work to the strict minimum. Sections 6 to 10 describe the facilities in detail through a set of increasingly ambitious examples (whose texts may all be found in the directory ***$EIFFEL3/examples/net*** of the Eiffel distribution). Section 11 is a short bibliography.

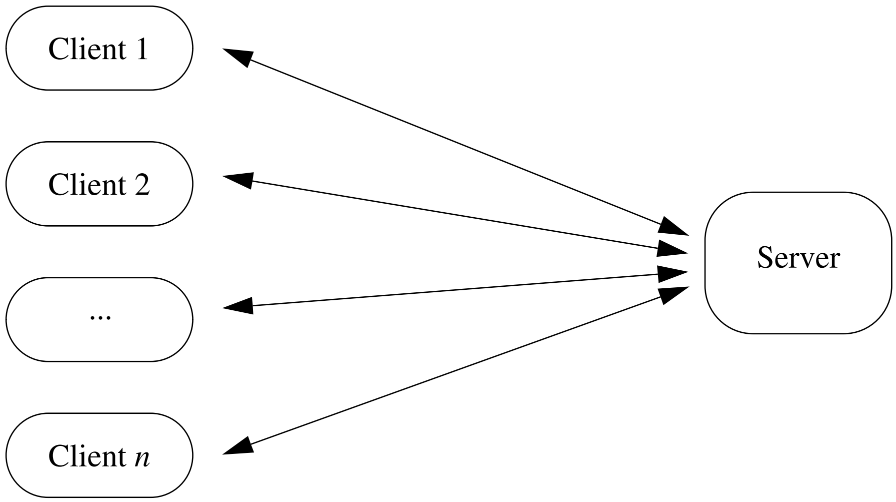
2 CLIENTS AND SERVERS

In the client-server model of computing, a number of software systems, the clients, may require general-purpose services provided by other systems, the servers. Clients and servers run concurrently, on the same machine or, more generally, on different machines connected through a network.

Typically, the clients are systems, for example systems that provide useful facilities for human users; the servers take care of operations that clients could not perform on their own (or would perform less efficiently), and of operations that are common to several clients. Example of servers include:

- **Computation servers**, which take care of heavy or specialized computations not appropriate for the clients, assuming the servers' processors are more powerful.
- **File servers** and **database servers**, providing access to persistent data that different clients may need to access.

The client-server relation is usually many-to-many: each client application, in the course of performing its duties, may successively require services from different servers; and each server may cater to different clients. This second aspect, illustrated by the following figure, is particularly important; it allows a single server to communicate with as many clients as appropriate.



Although communication occurs both ways — clients sending objects to servers and conversely — the situation in such a scheme is usually not symmetric. Whereas a client requesting a certain service will need to know beforehand what server provides that service, the server, for its part, will not in general name its clients in advance; it will simply stand ready to provide its services to whatever suitable client happens to request them.

This dissymmetry appears clearly in EiffelNet and in the examples of this manual. It means in particular that clients and servers act differently at the **beginning** of their client-server lifecycle. The party that initiates the communication is the client; in this operation it will identify the desired server (through a local file name for communication on the same machine, or though a machine name for network communication). The server simply makes itself ready for possible connections by starting to "listen" on the communication channels — relying for this on an EiffelNet procedure that is indeed called *listen*.

3 AN OVERVIEW OF EIFFELNET MECHANISMS

To enable clients and servers to exchange objects, you will have to ensure that they can refer to a common **address**. At the predefined level this is really the only notion that you need to know, although it is useful to get the bigger picture, in particular the concept of **socket** (which enables systems to set up communication channels), the various forms of communication (single-machine versus multi-machine, stream versus datagram), the kinds of object structure that may be exchanged, the notion of packet, and how to associate commands with communication events. The following paragraphs review these ideas and the corresponding EiffelNet abstractions.

3.1 Establishing a common address

When two systems need to communicate through sockets, they must establish a binding through some common point of reference, called an **address**. Predictably, the notion of address is one of the important internal abstractions of EiffelNet, although in most cases developers of applications using EiffelNet do not need to manipulate address objects directly.

EiffelNet supports two modes of communication: single-machine and multi-machine. In the single-machine case, the two communicating systems are known to be running on the same machine. In the multi-machine case, also known as the **network** case, they may be running on different machines, and communication occurs through a network. These two modes clearly require a different binding mechanism.

When a client and a server reside on the same machine, they both have access to that machine's file system. This provides a straightforward binding mechanism: the common address will simply be a "path name", the Unix terminology for the full name of a file in a hierarchically organized file system. In the examples below this file will be */tmp/here* (file *here* in the */tmp* directory, conventionally used for temporary files). This file must not exist prior to the communication; it will be created by the socket mechanisms and then removed.

For the network style of communication, this simple device of using a path name is no longer applicable. To define a common address, we may use two elements of information: the name of a machine, and the indication of a **port** on that machine. More precisely:

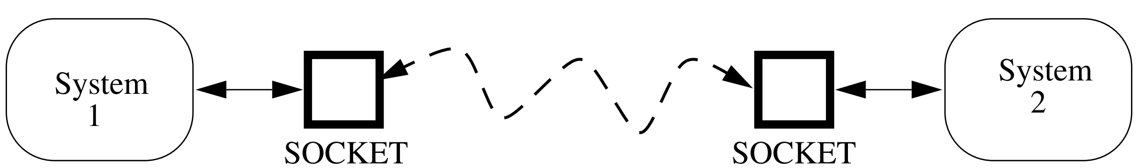
- The port will be identified by an integer. In the examples below port *2000* will be used.
- The machine may be identified in either of two ways: its **host name** (the name assigned to the machine when the operating system was first installed on it) or its **Internet address**, a sequence of numbers separated by periods, such as *127.0.0.1*. EiffelNet routines that need an argument identifying a machine will indifferently take a host name or an Internet address, passed in either case as a string. In the examples below the identification will be a host name, given as the string *"serverhost"*.

In network-style client-server communication, the mechanism will be dissymmetric, reflecting the possibility (noted earlier) of a single server catering to many clients. The clients will state both the machine identification of their intended server and the port on which they will talk to that server. The server, however, will only specify the port; this means that it makes itself available to any client that cares to talk to it on that port. This provides some of the essential flexibility of client-server communication, where only one of the partners needs to state beforehand whom it wants to talk to.

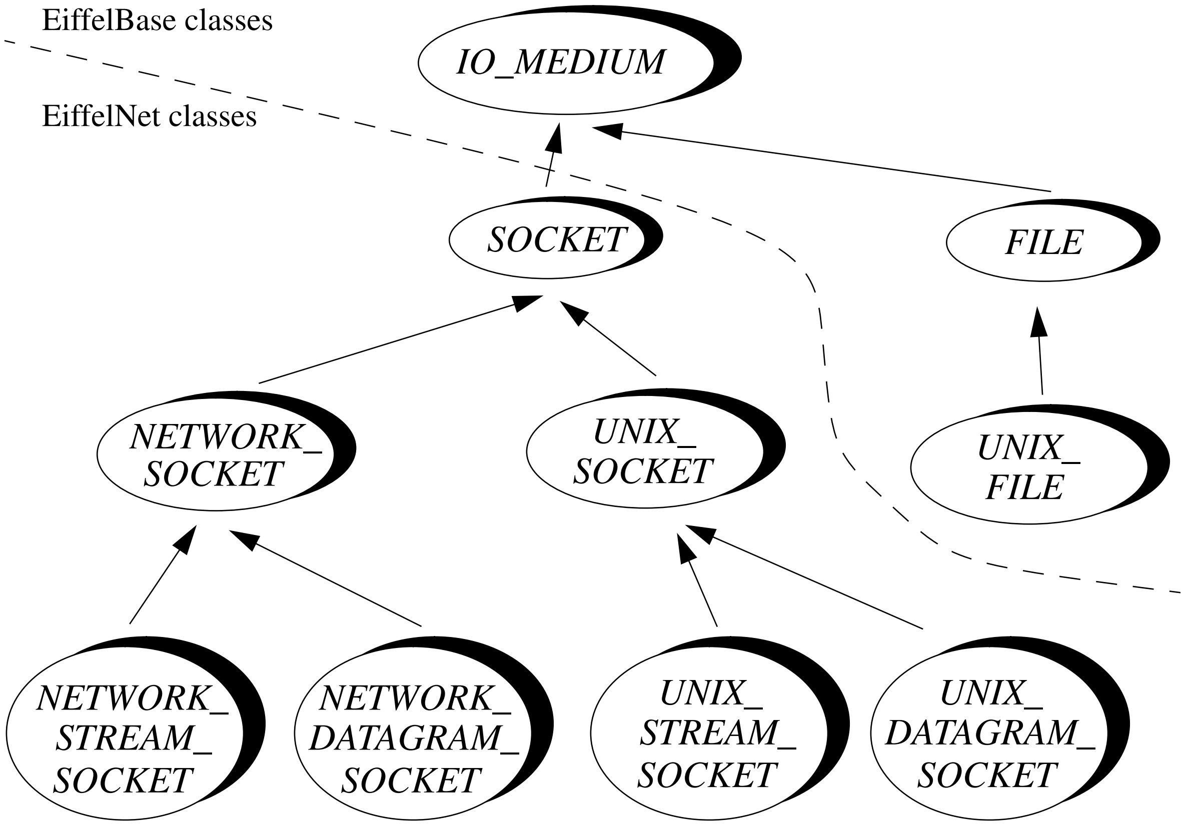
3.2 Sockets and communication modes

A software system will exchange objects with another by sending them to a socket. Although if you stay at the predefined level you will not need to manipulate sockets explicitly, it is useful to understand this concept and know about the corresponding EiffelNet classes.

You may think of a socket as a communication port; by attaching sockets together you enable communication between the corresponding systems, for example a client and a server:



EiffelNet has been designed so that sockets look very much like files. You send objects to a socket in the same way that you write objects onto a file, and receive objects from a socket in the same way that you read objects from a file. This fundamental commonality is reflected in the inheritance hierarchy of the corresponding classes:



Note that the hierarchy as shown is not complete; in particular the full structure uses classes *STREAM* (of which the *STREAM_* classes are heirs) and *DATAGRAM* for multiple inheritance. Only the classes below the dotted line are part of EiffelNet; the others are part of EiffelBase, the fundamental data structure and algorithm library of ISE Eiffel [2].

The most important property of this inheritance hierarchy is that it shows how sockets fit within the overall structure. Thanks to the common ancestor *IO_MEDIUM*, socket classes have most of their features in common with files.

In normal usage, the only socket classes that you will need are four classes appearing at the bottom of the above figure. They correspond to two separate distinctions: single-machine versus multi-machine, and reliable versus unreliable.

On the first distinction:

- If the communicating systems run on the same machine, you may use one of the *UNIX_* classes.

- For systems that run on different machines, you must use one of the *NETWORK_* classes. This will also work if the systems are on the same machine, but less efficiently since communication may go through the network.

The use of the word *UNIX_* does not mean that the machine must be running the Unix operating system; rather, it denotes a certain style of client-server communication, the Unix style. (This is comparable to the use of the name *UNIX_FILE* in EiffelBase, for a class describing files that behave in the Unix style even though they may be implemented on non-Unix machines.)

The second distinction reflects two modes of socket communication: stream communication and datagram communication. Both of these modes support two-way communication between systems, but with different properties:

- A stream socket, as provided by the *STREAM_* classes, provides sequenced communication without any loss or duplication of data. Stream communication is normally synchronous: the sending system waits until it has established a connection to the receiving system and transmitted the data.
- A datagram socket, as provided by the *DATAGRAM_* classes, is asynchronous: the sending system emits its data and does not wait for an acknowledgment. Because the sender is not blocked, this mode is more efficient, but it does not guarantee sequencing, reliability or non-duplication.

3.3 Sending and receiving simple values

IO_MEDIUM has all the basic input and output facilities applying to objects of basic types, as also offered in *FILE* (see the specification of *FILE* in reference [2]). So you can use sockets to send and receive characters, integers, real numbers in simple or double precision and strings. For example, if the type of *my_socket* is one of the socket classes shown on the preceding figures, any of the above calls will be valid:

my_socket.*putstring* (*"Some text"*)
my_socket.*readint*; *my_last_integer* := *my_socket*.*lastint*

Since sockets are bidirectional, these instructions may all appear as part of the same class provided you make sure to guarantee proper synchronization between senders and receivers. You may also prefer to specialize certain sockets for sending and others for receiving.

3.4 Sending and receiving object structures

In many cases, what you will want to send and receive is not just simple values but non-basic objects (instances of arbitrary classes, having as many fields as needed) and, more generally, entire object structures.

The basic mechanism enabling a system to send objects through EiffelNet is also the basic mechanism for storing objects into a file: class *STORABLE* from EiffelBase.

As documented in [2], *STORABLE* provides features to store and retrieve complete object structures. There are three storage procedures, called under the respective forms

struct1.*basic_store* (*iom1*)
struct1.*general_store* (*iom1*)
struct1.*independent_store* (*iom1*)

assuming that the type of *iom1* is *IO_MEDIUM* or a conforming type such as *FILE* or one of the *_SOCKET* classes, and that the type of *struct1* conforms to *STORABLE.*

Note that reference [2] in its original version does not include *independent_store*, and requires *iom* to be of type *FILE* rather than the more general *IO_MEDIUM*. The current version of EiffelBase, however, supports the more general properties described here.

All three storage procedures have the effect of sending to *iom1* (whether a file, a socket or some other IO-medium) a copy of the entire object structure starting at *struc1*. Together with the retrieval routines seen below, they apply the principle of reference completeness stated in [1] and [2]:

> Whenever a routine of class *STORABLE* stores an object into an external file, it stores with it the dependents of that object.
>
> Whenever one of the associated retrieval routines retrieves a previously stored object, it also retrieves all its dependents.

For EiffelNet, of course, “storing” and “retrieving” mean sending and receiving. The rest of this section, which applies to sockets as well as to files, will continue to use the original terminology of storage and retrieval.

The three storage procedures differ in their degree of generality:

- *basic_store* will only work if the sending and retrieving are performed by instances of the same system (the same executable module).
- *general_store* will work if the sender and retriever are different systems (using the same classes for the objects that they exchange), but these systems must run on machines of the same architecture, or at least of architectures that use the same data representation.
- *independent_store* will work in the most general case, with the sender and receiver possibly running on platforms using different data representations.

The penalty for using more general representations is that the data representation (as stored into the file or sent to the socket) will have to include more information. So *basic_store* uses the most compact representation, and *independent_store* the most verbose.

The scheme for accessing an object structure produced by one of these three procedures is the following, used in a descendant of class *STORABLE*:

struct2 ?= *retrieved* (*iom2*)

Here *iom2* must be of a type conforming to *IO_MEDIUM*. The assignment attempt ?= checks that the root object of the structure produced by the corresponding call to one of the *_store* procedures is of a type that conforms to the type of *struct2*; if not, the assignment will assign to *struct2* a void reference.

Although there are three separate storage procedures, there is only one retrieval routine, *retrieved*; the algorithm for *retrieved* is able to figure out, from the format of the retrieved objects, whether they were produced in the basic, general or independent mode.

3.5 Packets

The classes *PACKET* and *DATAGRAM_PACKET* are used to represent packets of data that can be sent to sockets.

Their main use is for a system that relies on datagram communication. As noted above, this mode does not guarantee sequencing, making each system responsible for checking that packets arrive in the proper order. This is possible through feature *number* of class *DATAGRAM_PACKET*, which gives the number of the current packet.

3.6 Associating commands with events

EiffelNet supports a highly asynchronous (and hence efficient) mode of operation by offering mechanisms through which you can specify that a certain action must be executed whenever a certain medium becomes available for reading, writing or handling of special cases (out of bounds). This facility is provided by a set of related classes:

- The actions are represented by class *POLL_COMMAND*, an heir of the EiffelBase class *COMMAND* with, in particular, the procedure *execute*.
- Using *MEDIUM_POLLER*, you can specify that a certain command (an instance of *POLL_COMMAND*) must be executed whenever a certain medium becomes available for the appropriate operation (read, write, handling of out-of-bounds cases).
- Using *POLL_MASK*, you can set a mask to select the sockets or files on which your instance of *MEDIUM_POLLER* is working.

4 THE PREDEFINED LEVEL

Many client-server applications follow a standard scheme that relies on the facilities outlined so far. When your application conforms to this scheme, you can avoid most of the work by relying on the client and server classes of the predefined level. There is no need to consider sockets explicitly or to worry about synchronization.

As in many other cases found in the ISE Eiffel 3 libraries, the object-oriented method makes it possible here to encapsulate the standard pattern found in most usual cases.

4.1 The predefined client and server classes.

The classes of the predefined level, which only support stream communication (not datagrams), are *CLIENT*, *SERVER*, and their descendants *UNIX_CLIENT*, *UNIX_SERVER*, *NETWORK_CLIENT* and *NETWORK_SERVER*. All the architecture of a client-server application is already available through these classes. All you have to do is to specify the address you want to use, what you want to send, when you want to send it, and what you want to do when you receive data. The rest will be taken care of automatically.

The *UNIX_* classes are meant for systems running on the same machine, using *UNIX_STREAM_SOCKET*; the *NETWORK_* classes support inter-machine communication, using *NETWORK_SOCKET*.

4.2 Using the client classes

When you want to write an application, your client should inherit from *UNIX_CLIENT* for example, and in your creation procedure, you have to setup your address, and then call the inherited make routine, with your address as an argument. You are then provided with some send and receive routines.

4.3 Using the server classes

For the server, you inherit for example from *UNIX_SERVER*. In the same way as for the client, you set up your address in the creation procedure, and then call the *make* creation procedure inherited from *UNIX_SERVER*. After that your creation procedure should call *execute*, which is inherited from *SERVER*, and provides you with a loop calling *receive*, *process_message*, *respond* and *clean_up*. Then, you should redefine these routines to do what you want your server to do.

5 INTRODUCTION TO THE EXAMPLES

The following sections describe a set of examples showing how to use EiffelNet to build client-server applications.

The examples start with the most simple uses involving high-level classes covering common cases and shielding developers from details of the mechanism. Subsequent examples will reveal some of these details, useful for more advanced or specific applications of EiffelNet.

All the examples discussed here appear in the directory ***$EIFFEL3/examples/net*** of the Eiffel distribution.

6 WORKING AT THE PREDEFINED LEVEL.

> The example classes discussed in this section appear in the subdirectory *predef* of the example directory

The first example shows how to use the predefined level classes, covering many common client-server schemes. The work needed in this case is minimal: just effect or redefine a few routines describing the specific processing that your client and your server require.

6.1 The example

The example describes a simple communication protocol. The client sends to the server a list of strings, which together form the message *"This is my test"*; the server receives the corresponding object structure, appends to it one more element — the string *"I'm back"* — and returns the result to the client, which will accept it and print it. The server increases a counter each time it closes a connection with a client, and will only accept three connections.

6.2 Object structures

As noted above, it is possible with sockets, as any other *IO_MEDIUM*, to send and receive simple objects such as integers. But for this first example we are already more ambitious and want to exchange entire linked lists of strings. The structures that we will exchange are described by the following class:

```
class OUR_MESSAGE inherit
    LINKED_LIST [STRING];
    STORABLE
creation
    make
end
```

Note that to make use of the storage and retrieval facilities the objects to be exchanged must be instances of a class which, as here, is a descendant of *STORABLE*.

6.3 Application classes and system structure

The client and server mechanisms will be described by instances of two classes that make up the example, called *OUR_CLIENT* and *OUR_SERVER*.

We want to have the two communicating systems be two different systems (compiled separately, with two separate Ace files). *OUR_CLIENT* will be part of one of these systems; *OUR_SERVER* will be part of the other.

There are four predefined classes, corresponding to the client and server sides with, in each case, the choice between the “Unix” (single-machine) and “network” (multi-machine) styles of communication. Note that all these classes support stream communication; for datagrams, see 8 , page 23. The server class should inherit from *UNIX_SERVER* or *NETWORK_SERVER*; the client class should inherit from *UNIX_CLIENT* or *NETWORK_CLIENT*. The choice between *UNIX_* and *NETWORK_* classes must of course be the same on both sides.

6.4 The client

Here is the client class:

```
class OUR_CLIENT inherit
    UNIX_CLIENT
        redefine received end
creation
    make_client
feature
    our_list: OUR_MESSAGE;
    received: OUR_MESSAGE; -- Type redefinition
```

```
        make_client is
                -- Build list, send it, receive modified list, and print it.
            do
                make ("/tmp/here");
                build_list;
                send (our_list);
                receive;
                process_received;
                cleanup
            rescue
                cleanup
            end;
        build_list is
                -- Build list of strings our_list for transmission to server.
            do
                !! our_list.make; our_list.extend ("This "); our_list.extend ("is ");
                our_list.extend ("our "); our_list.extend ("test")
            end
        process_received is
                -- Print the contents of received in sequence.
            do
                if received = Void then
                    io.putstring ("No list received")
                else
                    from received.start until received.after loop
                        io.putstring (received.item);
                        received.forth
                    end
                end;
                io.new_line
            end
    end
```

The scheme is very simple:

- The inherited procedure *make* takes care of the many details of creating a socket and setting up the communication; the following sections will give some insight into what this involves. The argument to *make* is the path name that has been selected to enable communication with the server: */tmp/here*.
- The client has defined a procedure *build_list* to construct an object structure.
- It then uses the inherited procedure *send* to send this object structure to the server; *send* expects an argument of a type conforming to *STORABLE*, so that it can use all the predefined storage and retrieval facilities.
- The *receive* procedure, called next, obtains an object structure from the server; here, as will be seen from the server's text below, this is the original list to which the server

has appended a new element. The result of procedure *receive* is available through feature *received*. To avoid the need for an assignment attempt (as with the result of a *retrieve* operation with *STORABLE*) the class redefines *received* to be of type *STORABLE*.

- The client has defined a procedure *process_received* to print the received string.
- Finally, a call to *cleanup* is necessary to close the sockets that have been opened and free the corresponding resources.

Of course, in a typical object-oriented design, the class and those which appear in the rest of this set of examples may "do" more than one thing and so may have more routines than the ones shown in this document.

This example shows how to write a client class in the most common cases: inherit from *UNIX_CLIENT* or *NETWORK_CLIENT*; use *make* to specify the socket address, *send* to send an object structure, *receive* to receive an object structure, *received* to access it, and *cleanup* to clean things up.

There would be one difference for a *NETWORK_CLIENT*: procedure *make* would need two arguments, the name of a host and a port on that host. For example, to connect to a server running on a machine of name *serverhost* and using port 2000, the call would be:

```
make ("serverhost", 2000)
```

As noted earlier, you may identify the machine by an Internet address such as *"127.0.0.1"* rather than by a host name.

6.5 The server

Now, the server class:

```
class OUR_SERVER inherit
    UNIX_SERVER
        redefine received, respond end
creation
    make_server
feature
    received: OUR_MESSAGE
    make_server is
            -- Receive list, print it, extend it, and send the result back.
        do
            make ("/tmp/here");
            execute
        rescue
            cleanup
        end;
```

```
process_message is
        -- Print received list.
    do
        from received.start until received.after loop
            io.putstring (received.item);
            received.forth
        end;
        io.new_line
    end;
respond is
        -- Extend received list.
    do
        received.extend ("%N I'm back.%N");
        resend (received)
    end
end
```

The developer's task is even easier here than on the client side. The last instruction in the creation procedure is *execute*, inherited from the chosen *_SERVER* class, which executes a loop that will repeatedly call *receive*, *process_message*, *respond* and *close*. This loop is actually an infinite one, which will only terminate if the server is killed (in which case a Rescue clause will execute procedure *cleanup*); indeed, a server is not expected to terminate under normal circumstances.

To define a server's behavior as desired for your particular application, all you are required to do is to effect the procedure *process_message*, inherited from the *_SERVER* class in deferred form. (Remember that to "effect" a deferred feature is to provide a non-deferred version, also called an effective version.) Here the effective version prints the list of strings received from the client.

In addition, you may also, as here, redefine procedure *respond*. In the *_SERVER* class this procedure is effective, but it does nothing; this means that you have nothing special to do for the common case in which the server does not respond when it has received a message. Here, however, we do want to respond; the redefined version of *respond* extends the *received* string and returns it. Procedure *resend* sends an object structure back to the last connected client. It is also possible to have more sophisticated versions of *respond*, for example to dispatch objects to all active clients; this will be shown in a later example

As in the client case there is a *cleanup* procedure to free the sockets, but you do not need to call it explicitly; it is called by *execute*. You may redefine this procedure to free some additional resources of your own.

7 OBTAINING A FINER DEGREE OF CONTROL

Let us now take a more internal look at the workings of EiffelNet. The two examples that follow have the same behavior as the preceding one; since their text is less simple, they are only interesting as an illustration of the lower-level facilities that you may want to use in specific cases. If you are already familiar with socket programming, they will also give you a more precise idea of how EiffelNet encapsulates the basic socket mechanisms.

As before, we have a client and a server class, still called *OUR_CLIENT* and *OUR_SERVER*, which are part of two different systems and will run concurrently. The communication uses streams rather than datagrams; the datagram form of communication will be examined in section 8 , page 23.

7.1 A client and a server on the same machine

> The example classes discussed in this section appear in the subdirectory *same_mach* of the example directory

First, let us assume that the client and server run on the same machine, so that we will use the *UNIX_* versions of the classes (the next example will be multi-machine). The communication protocol is also the same as before: the client sends a list of strings, the server returns it extended.

Here we will create and manipulate sockets directly. For that reason, both classes inherit from the EiffelNet class *SOCKET_RESOURCES* which introduces a number of constants and other useful socket-related features.

The two sockets must be able to refer to a common address. For a communication within a single machine, as noted, this address is a path name, again */tmp/here* for this example. The address will be an argument of the creation procedure used to obtain a socket *soc1* on either side:

```
!! soc1.make_client ("/tmp/here")
!! soc1.make_server ("/tmp/here")
```

The *make_* procedures take care of all the hassles of establishing a socket for a client or a server: creating an address object, setting it to the given path name, binding the socket to the address, and in the client case establishing the connection. For finer control, these procedures are still available: you can create a bare socket by using the basic creation procedure *make* (rather than the more sophisticated *make_client* and *make_server*), then create a separate address object, associate the two, and call the *bind* and *connect* procedures.

Because communication is bidirectional, the distinction between client and server is not between who sends and who receives, although here the server only sends messages of acknowledgment. The client is the party that initiates the communication; the server is the party which stands ready to accept the communication. This difference justifies the presence of two creation procedures *make_client* and *make_server* as illustrated above. To initiate the communication, the client will execute:

```
soc1.connect
```

To make itself ready for the communication, the server will execute:

```
soc1.listen (n)
```

where *n* is a positive integer indicating the maximum number of client connection attempts that may be queued. The value 5, used in the example, is typical for *n*.

When you use the *_SERVER* classes of the predefined level, as in the earlier example, 5 is indeed the default; you can change the value to a positive integer *n* through the call *set_queued* (*n*).

Whenever the server needs to exchange objects with one of the clients, it obtains access to the socket through the following sequence:

*soc1***.***accept*;

soc2 := *soc1***.***accepted*;

... Storage and retrieval operations using *soc2* (not *soc1*) ...;

*soc2***.***close*

Procedure *accept* ensures synchronization with the client. When communication is established, *accept* creates a new socket which will be accessible through attribute *accepted*, whose value is here assigned to the local entity *soc2*. To receive objects, the server will use operations of the form introduced earlier (3.4 , page 10):

struct ?= *soc2***.***retrieved*

applying to *soc2*, not *soc1*; this makes *soc1* available to accept connections with other clients, a fundamental feature of client-server mechanisms.

The operation *soc2***.***close* which terminates the above sequence closes the new socket. In principle this is not necessary, since garbage collection should eventually reclaim the socket object, and the *dispose* procedure of the corresponding socket class includes a call to *close*. But the risk exists that you run out of sockets before garbage collection reclaims all currently opened sockets, so it is preferable to include the *close* calls explicitly.

At the end of the processing it is necessary to close the original socket *soc1* but also to unlink it. The feature *cleanup* from class *SOCKET* takes care of both closing and unlinking.

Here is the server class based on these principles. The actual processing has been put aside in a procedure *process*.

class *OUR_SERVER* ***inherit***

SOCKET_RESOURCES;

STORABLE

creation

make

feature

soc1, soc2: *UNIX_STREAM_SOCKET*;

```
        make is
                -- Accept communication with client and exchange messages
            local
                count: INTEGER
            do
                !! soc1.make_server ("/tmp/here")
                from
                    soc1listen (5); count := 0
                until
                    count = 3
                loop
                    process; -- See below
                    count := count + 1
                end
                soc1.cleanup
            rescue
                soc1.cleanup
            end;
        process is
                -- Receive a message, extend it, and send it back
            local
                our_new_list: OUR_MESSAGE
            do
                soc1.accept;
                soc2 ?= soc1.accepted;
                our_new_list ?= retrieved (soc2);
                from our_new_list.start until our_new_list.after loop
                    io.putstring (our_new_list.item);
                    io.new_line;
                    our_new_list.forth
                end
                our_new_list.extend ("%N I'm back.%N");
                our_new_list.general_store (soc2);
                soc2.close
            end
    end
```

Note that at the end the server should not only closes the original socket *soc1* but also unlinks it. It is recommended to have a Rescue clause which, as here, ensures that the socket will be closed and unlinked if the system terminates abnormally before its term.

Here now is the client class:

```
class OUR_CLIENT inherit
    SOCKET_RESOURCES
creation
    make
feature
    soc1: UNIX_STREAM_SOCKET;
    make is
            -- Establish communication with server, and exchange messages
        do
            !! soc1.make_client ("/tmp/here");
            soc1.connect;
            process; -- See below
            soc1.cleanup
        rescue
            soc1.cleanup
        end
    process is
            -- Build a message to server, receive answer, build
            -- modified message from that answer, and print it.
        local
            our_list, our_new_list: OUR_MESSAGE
        do
            !! our_list.make;
            our_list.extend("This"); our_list.extend (" is");
            our_list.extend (" our"); our_list.extend (" test");
            our_list.general_store (soc1);
            our_new_list ?= soc1.retrieved;
            from
                our_new_list.start
            until
                our_new_list.after
            loop
                io.putstring (our_new_list.item);
                our_new_list.forth
            end;
            io.new_line
        end
end
```

7.2 Communication between two different machines

> The example classes discussed in this section appear in the subdirectory *two_mach* of the example directory

Let us now assume that the client and the server will run on two separate machines. Instead of *UNIX_* sockets, we must now use sockets of type *NETWORK_STREAM_SOCKET*.

The available creation procedures are slightly different. The server will be set up so as to listen to clients from any machine; it designates a **port**, identified by an integer, on which it will listen. The socket creation on the server side is then

```
!! soc1.make_server_port (2000)
```

For the client, the creation will specify two elements of information: the port number and the server. The server argument, a string, identifies the machine used as a server; it may be the host name of that machine, for example *"serverhost"* as used in the example; or it can be the machine's internet address, made of a sequence of numbers separated by periods, such as *"127.0.0.1"*.

The rest of the classes is as before.

```
class OUR_SERVER inherit
    SOCKET_RESOURCES
    STORABLE
creation
    make
feature

    soc1, soc2: NETWORK_STREAM_SOCKET;
    make is
            -- Accept communication with client and exchange messages
        do
            !! soc1.make_server_by_port (2000);
            ... The rest as before...
        end
    process is ... As before ...
end

class OUR_CLIENT inherit
    SOCKET_RESOURCES
creation
    make
feature
    soc1: NETWORK_STREAM_SOCKET;
```

```
        make is
            do
                !! soc1.make_client_by_port (2000, "serverhost");
                ... The rest as before ...
            end
        process is ... As before ...
    end
```

8 USING DATAGRAM SOCKETS

> The example classes discussed in this section appear in the subdirectory *datagram* of the example directory

Using datagram sockets implies a few differences from the preceding schemes. You create a *NETWORK_DATAGRAM_SOCKET* or a *UNIX_DATAGRAM_SOCKET* as before. You do not need to call *listen*, *accept* or *connect*. Once you have created your sockets using the *make_client* or *make_server* creation procedure, you can use the *send* and *received* routines, specifying the socket to which you want to send, or from which you want to receive.

The command *send* takes an argument of type *DATAGRAM_PACKET*, and the query *received* returns a result of this type. *DATAGRAM_PACKET* is an heir of class *PACKET*, itself an heir of *ARRAY* [*CHARACTER*]. Class *DATAGRAM_PACKET* makes it possible, through the query *number* and the associated command *set_number*, to associate a packet number with each packet. This compensates for lack of guaranteed sequencing of datagram sockets: the sender can number packets through *set_number*, and the receiver can check that all packets have been received through *number*, asking the sender to re-emit missing objects.

The text of this example is similar to what appears below (in a more general setting involving event-driven computation) for the next example.

9 EVENT-DRIVEN COMMAND EXECUTION

> The example classes discussed in this section appear in the subdirectory *polling* of the example directory

9.1 Commands and events

In the preceding examples each participant in a communication had to get ready to send or receive at specific stages of its life. Although this did not preclude asynchronous communication, it is sometimes desirable to make the scheme even more asynchronous, and control more decentralized, by letting each system simply specify certain communication events that it wants to monitor, and certain commands to be executed on occurrence of the specified events.

This event-command model of computation is of course not new with EiffelNet. In ISE Eiffel 3, it is already used in EiffelVision and especially in EiffelBuild, which makes it possible to build a graphical application by specifying which command should be executed in response to each of a set of specified interface events, such as mouse click or keyboard entry. The commands are objects, instances of a general-purpose class *COMMAND* or its proper descendants. Class *COMMAND* has, among its features, a procedure *execute* which executes the current command; some commands are undoable and have an *undo* procedure.

Class *COMMAND* and related classes used to be in EiffelVision but because of their generality were moved to [2], although this occurred after the publication of [2].

In EiffelNet the possible events associated with a socket will be of three kind: a read event; a write event; or a special event (out of bounds operation). The command classes will be descendants of *POLL_COMMAND*, an heir of *COMMAND*.

9.2 Command classes

The example uses four command classes: *CLIENT_DATAGRAM_READER*, *CLIENT_DATAGRAM_WRITER* and their counterpart for servers, representing operations that must be triggered in the case of a read event and a write event.

Here is the reader command for clients:

```
class CLIENT_DATAGRAM_READER inherit
    POLL_COMMAND
        redefine active_medium end
creation
    make
feature
    active_medium: NETWORK_DATAGRAM_SOCKET;
    execute is
                -- Obtain a packet of ten characters and print them.
        local
            rec_pack: DATAGRAM_PACKET; i: INTEGER
        do
            rec_pack := active_medium.received (10, 0);
            io.putint (rec_pack.packet_number); io.new_line;
            from i := 0 until i > 9 loop
                io.putchar (rec_pack.element (i)); i := i + 1
            end;
            io.new_line
        end
end
```

The *execute* procedure reads a packet of ten characters and prints these characters. Its counterpart in the writing command will produce these ten packets:

```
class CLIENT_DATAGRAM_WRITER inherit
    POLL_COMMAND
        redefine active_medium end;
    BASIC_ROUTINES
creation
    make
feature
    active_medium: NETWORK_DATAGRAM_SOCKET;
    execute is
            -- Make a packet with characters 'a' to 'k' in successive positions.
        local
            sen_pack: DATAGRAM_PACKET; ccode: INTEGER
        do
            !! sen_pack.make (10);
            from ccode := charcode ('a') until ccode > charcode ('k') loop
                sen_pack.put_element (charconv (ccode), ccode – charcode ('a'));
                ccode := ccode + 1
            end
            sen_pack.set_packet_number (1);
            active_medium.send (sen_pack, Void, 0)
        end
end
```

9.3 The server and the client

Once the commands have been defined, it suffices for the server and the client to associate instances of these commands with the appropriate.

The abstraction needed for this purpose is provided by class *MEDIUM_POLLER*. An instance of this class knows about a number of commands, each associated with a certain socket in read, write or special event mode. By applying procedure *execute* to such a medium poller, you direct it to monitor these sockets for the corresponding events and to execute the command associated with each event that will be received. Procedure *execute* takes two integer arguments: the maximum number of sockets to monitor, and the timeout in milliseconds.

Here is the server built with this mechanism:

```
class POLLING_SERVER creation
    make
feature
    make is
            -- Create read and write commands, attach them to a poller,
            -- set up the poller for execution.
        local
            soc: NETWORK_DATAGRAM_SOCKET;
            poller: MEDIUM_POLLER;
            readcomm: SERVER_DATAGRAM_READER;
            writecomm: SERVER_DATAGRAM_WRITER
        do
            !! soc.make_server_by_port (6530);
            !! poller.make;
            !! readcomm.make (soc); poller.put_read_command (readcomm);
            !! writecomm.make (soc); poller.put_write_command (writecomm);
            poller.make_read_only; poller.execute (15, 20000);
            poller.make_write_only; poller.execute (15, 20000);
            soc.close
        end
end
```

Procedure *make* creates three objects: a socket, which it associates with a specific port; a poller; and a read command (an instance of *SERVER_DATAGRAM_READER*), which it attaches to the socket. It then enters the read command into the poller, and does the same thing with a write command. It sets up the poller to accept read commands only and then executes the poller; this will enable the server to get the read event triggered by the client's write command (as it appears below in the text of class *POLLING_CLIENT*). Then the server reverses the poller's set-up to write-only, and calls *execute* again.

The procedures *make_read_only* and *make_write_only* are creation procedures, so that it is possible in a single instruction to create a poller and set it up for read-only or write-only, as in !! *poller*.*make_read_only*. For clarity, however, the above class and the next separate calls to these procedures from the creation of the poller, which uses *make* as creation procedure.

The client follows the same scheme, reversing the order of read and write operations:

```
class POLLING_CLIENT creation
    make
feature
    make is
            -- Create read and write commands, attach them to a poller,
            -- set up the poller for execution.
        local
            soc: NETWORK_DATAGRAM_SOCKET;
            poller: MEDIUM_POLLER;
            readcomm: DATAGRAM_READER;
            writecomm: DATAGRAM_WRITER
```

```
        do
            !! soc.make_client_by_port (6530, "serverhost");
            !! poller.make;
            !! readcomm.make (soc); poller.put_read_command (readcomm);
            !! writecomm.make (soc); poller.put_write_command (writecomm);
            poller.make_write_only; poller.execute (15, 20000);
            poller.make_read_only; poller.execute (15, 20000);
            soc.close
        rescue
            if soc /= Void and then not soc.is_closed then
                soc.close
            end
        end
    end
```

9.4 A less deterministic scheme

Although the example uses the event-driven mechanisms of EiffelNet, it is still relatively deterministic in that it follows a precise protocol defined by a strict sequence of read and write operations on both sides. This is why every call to *execute* is preceded by a call to either *make_read_only* or *make_write_only* to set up the poller in the appropriate mode.

A less deterministic scheme may often be desirable, where you simply enter a number of commands (read, write, out of bounds processing) into a poller and then wait for arbitrary events to occur and trigger commands. There is no need with this scheme to know in advance the order in which events may occur: a read event will trigger the command entered into the poller through *put_read_command*; a write event will trigger the command entered through *put_write_command*.

To achieve this behavior, simply create the poller using *make* as creation procedure. This will set up the poller so as to accept all socket events, and enter into event-driven command execution by calling *execute* on the poller.

10 A MORE COMPLEX EXAMPLE

> The example classes discussed in this section appear in the subdirectory *advanced* of the example directory

The last example exercises most of EiffelNet's major facilities. It consists of a server that allows an arbitrary number of clients to connect to it. Each time the user of one of the client systems types a line on the keyboard, the client sends this character to the server, which then broadcasts it to all the clients (including the original sender). This scheme allows several people to talk together, hence the names chosen: the server class is called *CHAT*, and the client is called *JOIN*.

The example uses the network mode of communication, based on the *NETWORK_CLIENT* and *NETWORK_SERVER* classes. It uses automatic polling through *MEDIUM_POLLER* as in the previous example; the relevant command is given by class *CONNECTION*, an heir of *POLL_COMMAND*. The information exchanged between the server and its clients is described by class *MESSAGE*, an heir of *LINKED_LIST* [*STRING*] similar to the earlier examples' *OUR_MESSAGE* (see see "Object structures", 6.2 , page 14). Attributes include, the name *client_name* of the client that has sent this message, the boolean *new* indicating whether the current message is the first from a client that is trying to connect to the server, and *over* indicating that the message is the last sent by a client before disconnecting.

The server maintains a list of the currently active connections. In the *receive* routine, it checks on the main socket for any client trying to connect. The socket is set to be non-blocking to enable the server to continue checking the already connected clients. If the connection is successful, the server sends to the new client the list of clients already connected and adds the new connection to its list. Then it polls the connections in the list, and processes the messages, if any. If the message is tagged *new*, the server sends a message to all the clients indicating that a new client has joined the server; if it is tagged *over*, it sends a message indicating that the client has opted out.

Each client uses the *MEDIUM_POLLER* to check any message coming from the server and immediately displays any such message. It also checks a special connection, created with *io*.*input* as a medium, to check what the user is typing and then send it to the server. If the user types *bye*, the client terminates, sending a message tagged *over* to the server.

11 BIBLIOGRAPHY

[1] Bertrand Meyer: *Eiffel*: *The Language*, Prentice Hall, 1992. (Available from ISE.)

[2] --: *Reusable Software*: *The Base Object-Oriented Component Libraries*, Prentice Hall, 1994. (Available from ISE and included in the standard delivery of ISE Eiffel.)

[3] W. Richard Stevens: *Unix Network Programming*, Prentice Hall, 1990.