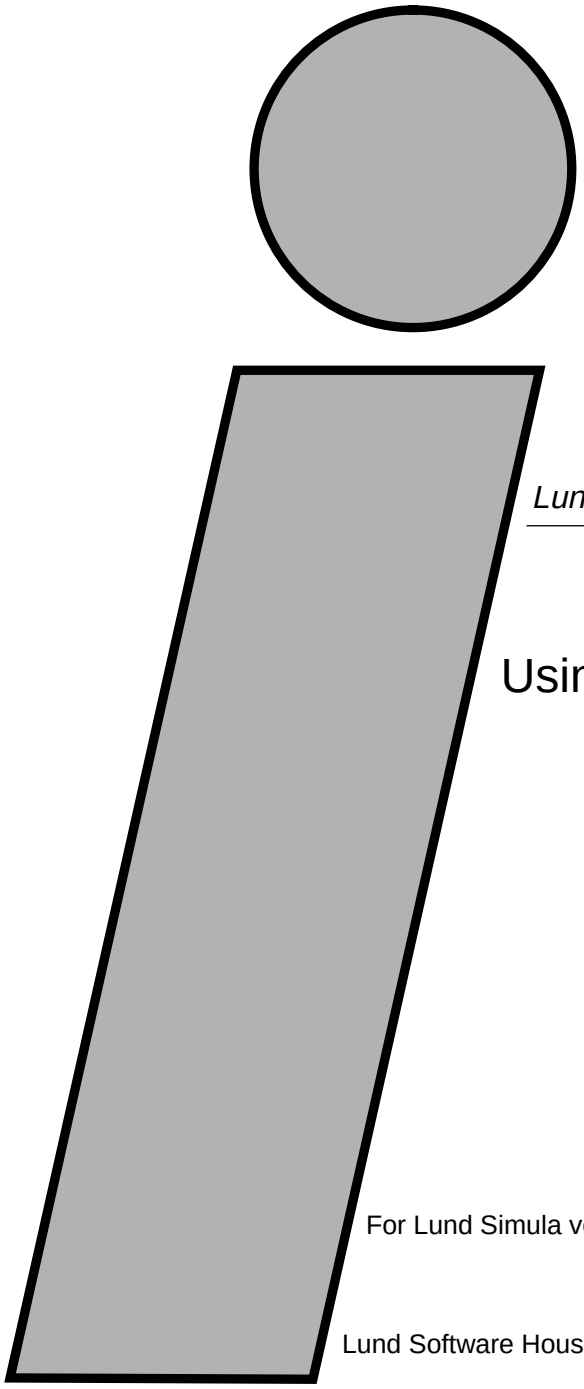


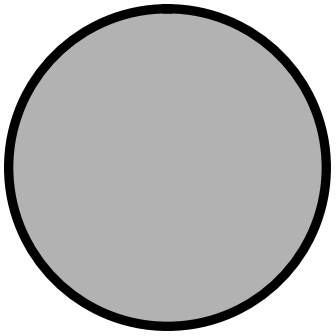
LUND

Lund Simula Documentation

Using the **simsocket** library on Unix Systems

For Lund Simula version 4.15 or later

Lund Software House AB, Sweden



SIMULA

Lund Simula Documentation

Using the SimSocket library on Unix Systems
Version 4.15

by Boris Magnusson

Printed at: 5 December 1995 2:00 pm

Lund Software House AB
P.O.Box 7056
S-220 07 Lund, Sweden

Table of Contents

1 Introduction 3

2 Organization and use 3

3 Overview of functionality 3
- 3.1 Establishing a connection 4
- 3.2 Client-Server Communication 4
- 3.3 Addressing 5

4 Abstract class descriptions 5
- 4.1 Class Hierarchy 5
- 4.2 class InternetAddress 5
- 4.3 classSocketBasics 7
- 4.4 class SocketIO 7
- 4.5 class ClientSocket 7
- 4.6 class ServerSocket 7
- 4.7 class ServerSwitch 8

5 Examples 8
- 5.1 Single customer Server 8
- 5.2 Simple Client program 9
- 5.3 Multiple customer Server 10

6 Detailed Interfaces 13
- 6.1 InternetAddress 13
- 6.2 SocketBasics 14
- 6.3 SocketIO 15
- 6.4 ClientSocket 16
- 6.5 ServerSocket 17
- 6.6 ServerSwitch 17

7 Index to classes and procedures 19

1 Introduction

'Socket' is an abstraction for communication between operating system processes, such as Simula programs. Two such communicating processes may be executing on the same compter, or on different computers connected via a network, possibly at a large geographical distance. Influenced by the role such communicating processes play they are often called Server and Client respectively. The purpose of this package is to make implementation of interprocess communication easy in Simula.

Communicating processes are useful in many situations. One typical situation, the database case, is to protect common data (on file) from simultaneous access. By letting only one process (a Server) do the access, it can serialize the access from many different concurrent (Client) processes. The Server can also guarantee that access and updates of the shared data is done in meaningful chunks (atomic operations) so no inconsistent, intermediate, states are visible to other Client processes. A slightly different example where communicating processes are useful is when resources on one machine is needed by users on other machines. Examples of this is the set of ftp and telnet programs. Such systems are usually organized as clients executing on the users machine and server-processes ('daemons' such as ftpd, telnetd) executing on remote machines.

2 Organization and use

The routines are distributed as separately compiled Simula classes. The interface files (*.atr-files) are normally installed in '/usr/local/simulabin/simsocket' and the object files in '/usr/local/lib/libsimsocket.a'.

Declaration in a Simula program:

external class ClientSocket;

Compilation:

% **simcomp <program> -L=/usr/local/simulabin -I=simsocket**

(or just: **simcomp <program> -I**)

Linking:

% **simld <program> -lsimsocket -lsocket -lnsl**

(or just: **simld <program> -l)**

See also BitPackClass, in the SimLib library, for outputing binary numbers and the SimIOProcess library for real-time applications in Simula.

3 Overview of functionality

Socket lends their name from the metaphor that they represent the endpoints in a communication wire with two ends. Information can be inserted (written) in one end and then (possibly a moment later) received (read) in the other end. The metaphor is slightly misleading since Sockets represents a double duplex connection, much like a phone-line, i.e. two pipes and two sets of connectors. This means that process A can write into its Socket at the same time as process B write into its socket, at the other end of the communication link. A moment later they can read the information written by the other process. In short there is at this level no presumed synchronization mechanism between the processes. In practice the communicating processes often behave according to some agreed protocol, one process is acting and responding as result of messages sent by the other process.

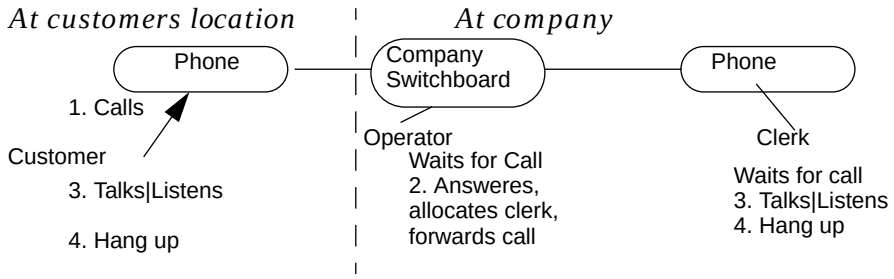


Figure 1 *Connection scenarios using a company switch-board*

3.1 Establishing a connection

The way a connection is established with Sockets is analogous to setting up a phone-call with a company (fig. 1). The customer takes the initiative, takes his phone and dials the number of the company, and the call reaches the exchange. The exchange allocates a clerk and the customer and the clerk starts exchanging messages. When one of the two hangs up the phone the connection is terminated and the other partner hangs up as well.

In the Socket case (see fig 2) the above scenario corresponds to a Client program using a ClientSocket and an InternetAddress to 'call' the Server program. An object of class ServerSwitch notices the incoming call and the connection is handed over to an object of class ServerSocket which can then be used for the message exchange.

Servers are typically intended to be able to serve several Clients at the same time. In order to support many simultaneous connection in a Server the mechanisms in the library SimIOProcess are useful. Typically one process is managing the ServerSwitch, waiting for incoming requests. When one arrives, this process will create and start another process, designed to service one connection. This service process will be handed a ServerSocket object and act as the clerk in the telephone exchange analog. A slight difference is that in this situation we create new Service processes at need, while in a company the same Clerk normally handles many calls after each other.

3.2 Client-Server Communication

When connection has been established the Client will have an object of class ClientSocket and the Server one of class ServerSocket. These objects act as a kind of files and support operations for sending and receiving messages, with operations such as Read and Write. The format of the information exchanged is text strings. Text strings can thus be conveniently sent, but other information such as integers or reals has to be formatted into a text buffer before sent. This can

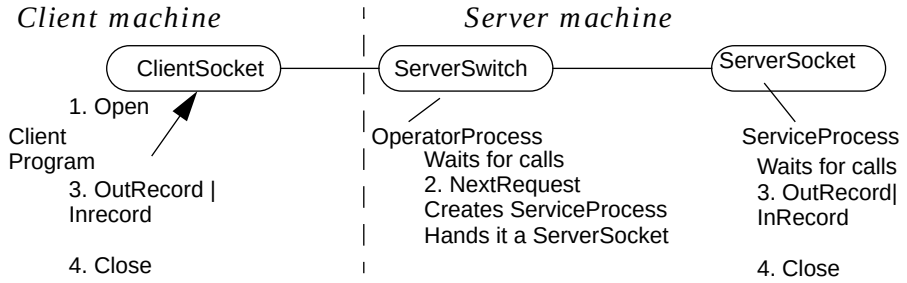


Figure 2 *Connection scenarios in Client-Server situation*

be done either as edited numbers, like when using an Outfile, or in binary form by use of the operations in BitPackClass of the LibSim library. When the information is read at the other end of the Socket, the receiver needs to know the format of the information read. It is thus an important part of implementing a Server to define a format for the messages it can receive and the messages it sends as replies. The implementation of a Client thus has to follow this Server specific message format.

ServerSocket and ClientSocket also has Close operations used to terminate a connection. When called the partner in the other end will notice this situation since the procedure EndFile will return true, very much as when a diskfile has been read to its end.

3.3 Addressing

Addresses of Sockets consists of three parts:

- Internet address – the 'Name' of the other machine.
- Port number – the 'Name' of the program on the other machine.
- Protocol – the 'Language' it speaks.

In order for a Client to communicate with a Server program it must know all these three aspects of the Server. Setting up a Server one has to define the Port number and the Protocol it understand. The machine aspect is given by the machine on which the Server program actually executes. The class InternetAddress abstracts all the three aspects above.

4 Abstract class descriptions

4.1 Class Hierarchy

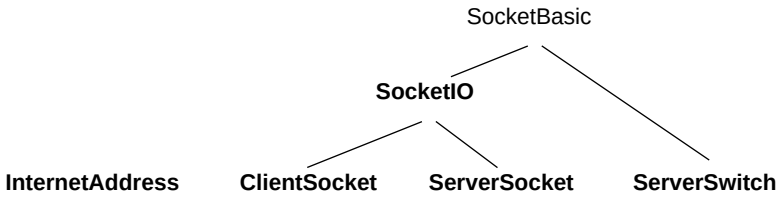


Figure 3 *Class hierarchy in SimSocket*

4.2 class InternetAddress

InternetAddress is providing all the mechanisms needed to define the other end of a Socket. A Client must give all the three aspects of a Socket in order to connect to a Server.

Internet address
Machines on Internet are really named by 32-bit integers, but there are several ”synonyms” for this unique number. The class InternetAddress implements various ways to define this 32-bit number:

- directly (in the rare case you already know it as a 32-bit integer)
- from the frequently used format of four integers: x.y.z.w
- from an alias name in text form as: ”biur.dna.lth.se” which is looked up in a translation table. This is the most commonly used form.

The class InternetAddress has operations to set the address in one of these forms and retrieving the address in the other forms. Clients usually define the machine to connect to in the textual form. Servers use this class to convert an Internet address of a connected Client to the textual form for logging, authorization etc.

Port Numbers
Port numbers are also integers defined by the Server program. The simplest way is that both Server and Client knows about the number. An alternative is to use a translation method where a table on the server machine is consulted to do the translate between an Mnemonic name and the number. This method has the advantage that it helps avoiding the situation that programs on the same machine use the same Port number, and also that a Client tries to use a Port number that is used by some other Server program on that machine than the expected. There is, however, still the possibility that the same mnemonic is used by different servers which can result in the same confusion. This implementation currently only support the direct numbering approach. There are conventions for how Port numbers are selected, some range of numbers are reserved. Check the file /etc/services to see which port numbers are actually used on your machine and to avoid conflicts.

Protocols
Protocols are organized in ‘Domains’. Currently this implementation only supports the Domain AF_INET. In this Domain there are two relevant Internet protocols:

- Stream/TCP
- DGram/UDP

Stream/TCP is providing a sequenced, reliable, two-way connection of bytes. Bytes sent are received in the order they are sent. Bytes sent in different write request can be concatenated (or split up) when read. The Stream implementation will send a constant trickle of small messages to keep the connection ‘warm’, in order to report connection failures (like Server or network failures).

DGram/UDP is a more primitive (but more efficient?) communication protocol. Messages are sent as fixed-size maximum length packets. Packets can be dropped, duplicated, received out of order etc., but when received they are read as a complete message.The user program has to take care of these problems, resending messages etc.

This implementation currently only support the Stream/TCP protocol.

4.3 classSocketBasics

This abstract superclass provides the Open/Close operations for its subclasses. Here is also protected interfacesfor Socket-related C-routines used to implement its subclasses.

4.4 class SocketIO

This class implements input and output operations on sockets, used in both Client and Server ends. There is a primitive I/O pair Read/Write, just shipping bytes, and a slightly more sophisticated pair of operations: InRecord/OutRecord that communicates variable size Texts over the pipe.

The Read/Write operations are directly calling the I/O operations provided for Sockets. They are thus not supporting any structure on the communication, but this has to be implemented by the application. One Write can be matched by many Reads (using smaller buffers) or several Writes can be matched by one Read (e.g. if the reader has been delayed). The Read/Write operations can be used to write Simula programs communicating with existing servers or clients.

In many cases the communication is structured as an exchange of messages. OutRecord/InRecord supports this organization. One call to OutRecord on the sending side always match one call of InRecord on the receiving side. These operations use a specific format of the messages and are only useful if there are Simula programs in both ends (or if a program written in another language can be adapted to use the same format).

A call to Read or InRecord will hang if there are no data available. The functionality provided by the library SimIOProcess can be used to avoid calling these operations until data is actually present (and thus avoiding the possible hang). See this library and the examples below for details.

SocketIO is an abstract class with two concrete subclasses: ClientSocket and ServerSocket.

4.5 class ClientSocket

This subclass of SocketIO is instantiated by a Client program. It is in many ways similar to a Simula file object, with Open, Endfile and Close operations. ClientSocket objects has, however, an object of class InternetAddress as a parameter rather than a textual file-name. The 'Open' operation is implementing the elaborate sequence of operations needed to establish a connection with a a Server. Using the TCP/Stream protocol this involves exchange of several messages and a time-out in case the external Server does not respond. The time to perform an Open operation is thus unpredictable. Operations for I/O are those described in SocketIO.

4.6 class ServerSocket

This subclass of SocketIO is instantiated by the Server program, one for each connected Client. When connected there are thus a one-to-one correspondence between a ServerSocket object and a ClientSocket object.

ServerSocket object is also similar to a Simula file object. The address of the ClientSocket object it will be connected to is, however, not known beforehand since the Client will take the initiative to call the Server. The 'Open' operation is usually not called directly by the application program but is called by the Nex-

tRequest operation in SocketService. As a result the address of the ServerSocket is defined (as an InternetAddress object denoting the connected Client machine). Also notice that if Sockets are not propperly closed, say if the program is terminated due to some error, it takes a while (some minutes) for Unix to realize that processes using the Socket are gone. In the meantime it is not possible to start a new Server using the same Port number. Operations for I/O are those described in SocketIO.

4.7 class ServerSwitch

There is usually only one object of this class in a Server where it defines a Port with a Protocol. It receives requests from Clients who wants to connect to the Server. This is handled by the operation NextRequest which will hang until a request is present. When this happens it initializes an object of class ServerSocket to be used to service the connection. If a Server defines several Ports (maybe providing different protocols or functionality) then there will several objects of class ServerSwitch.

The functionality provided by the library SimIOProcess can be used to avoid calling NextRequest until a request is actually present (and thus avoiding the possible hang). See this library and the examples below for details.

5 Examples

5.1 Single customer Server

This Server is a very simplified implementation of a global 'copy-paste' buffer. Client programs can send strings and the Server will remember the last string sent. Clients can also ask for the last string. This is thus a mechanism for communication between programs that do not know each other.

This first version of a server can only service one external client (customer) at the time. I/O read operations such as Inrecord and NextRequest will hang until data is avialable. See example 5.3 for how to implement a server that can handle many clients in parallel.

The format of the information the Server understands is:

'P'<message> and 'G'

if first charcter is a 'P', the the rest of the message is the text to store.

if the first char is a 'G', the message is only one character.

replies sent by the Server are:

command 'P' – "OK"

command 'G' – the stored text.

In all other situations, the server replies with a single character message containing "?".

```
! ------------------------------------------ file: singleserver.sim;
begin
   External class InternetAddress, ServerSwitch, ServerSocket;
   ref(InternetAddress) Us;
   ref(ServerSwitch) Switchboard;
   ref(ServerSocket) Customer;
   text T,Remember;
   character CMD, Get='G', Put='P';
```

```
    Us:- new InternetAddress;
    Us.SetPort(11475); Us.SetStream;
    Switchboard:-new ServerSwitch(Us);
    if not Switchboard.open(4) then
       Error("Can not create server socket");
    T:- blanks(80);
    while true do
    begin
       Customer:-Switchboard.NextRequest(new ServerSocket); ! Wait for call;
       outtext("New Call from Client: ");
       Outtext(Customer.GetClientAddress.HostName); outimage;
       T:-Customer.InRecord(T); ! Wait for him to say something ;
       while not Customer.EndFile do
       begin
          CMD:=T.getchar;
          if CMD=Put then
          begin
             Remember:-Copy(T.sub(2,T.length-1).strip);
             Customer.outrecord("OK");
          end
          else if CMD=Get then
             Customer.outrecord(Remember)
          else
             Customer.outrecord("?");
          T:-Customer.InRecord(T);
       end;
       Customer.Close;
       outtext("Client hang up"); outimage;
    end - while -;
end
```

5.2 Simple Client program

This simple Client program can be used to test the Server. It reads a line from the keyboard, sends it to the Sever, waits for the reply and prints it to the screen. The interactive user must thus know the message format understod by the Server.

```
! ------------------------------------------ file: simpleclient.sim;
begin
   External class ClientSocket;
   External class InternetAddress;

   ref(ClientSocket) Company;
   ref(InternetAddress) CompanyNumber;
   text T,Inbuff;

   CompanyNumber:-new InternetAddress;
      !-- "localhost" means the same machine: --;
   CompanyNumber.SetHostByName("localhost");
   CompanyNumber.SetPort(11475);
   CompanyNumber.SetStream;
   Company:-new ClientSocket(CompanyNumber);
   if not Company.open then
```

```
        Error("Server not responding");
    Sysin.Inimage;
    while not Sysin.Endfile and not Company.Endfile do
    begin
        T:-Sysin.Image.Strip;
        Company.Outrecord(t);
        Inbuff:-Company.InRecord(Inbuff); ! Waits for response ;
        if Company.endfile then
            Error("Server hang up")
        else
        begin
            Outtext("Response: "); Outtext(inbuff); outimage;
            Sysin.Inimage;
        end;
    end;
    Company.Close;
end
```

5.3 Multiple customer Server

This implementation of a Server demonstrates how the SimIOProcess library can be used to write real-time programs in Simula. This Server can simultaneously handle many Client programs connected over sockets. The messages from the Clients will be handled one at the time as they arrive. The Server also have a process that can respond to commands typed at the keyboard on the Server machine. One command (ctrl-D) will close down the Server. This example has also been implemented as a set of separately compiled classes and a main program. The message format used by this Server is the same as described in the first example 5.1.

```
! ------------------------------------------------- file: memory.sim;
External class Monitor;
Monitor class Memory;
begin Text Remember;
    procedure Put(t); Text t;
    begin EnterMonitor; Remember:-t; ExitMonitor;
    end - Put -;
    text procedure Get;
    begin EnterMonitor; Get:-Remember; ExitMonitor;
    end - Get -;
end -- Memory --;
! ------------------------------------------------- file: clerkprocess.sim;
External class IOProcess, ServerSocket, Memory;
IOProcess class ClerkProcess(toClient,M);
    ref(ServerSocket) toClient; ref(Memory) M;
begin
    ref(ProcessEvent) Event;
    text T,Inbuff;
    character CMD, Put='P', Get='G';
    EnableInputEventsUNIX(toClient.GetChannelNo);
    EnableCancelEvents;
    while not Event in CancelEvent and not toClient.EndFile do
    begin
        Event:-WaitEvent; ! ---ClerkProcess waits --;
```

```
        inspect Event ! -- ClerkProcess continues --;
        when InputEvent do
        begin
           T:-toClient.InRecord(T);
           if not toClient.Endfile then
           begin
              CMD:=if T.more then T.getchar else '?';
              if CMD=Put then
              begin
                 M.Put(Copy(T.sub(2,T.length-1).strip) );
                 toClient.outrecord("OK");
              end
              else if CMD=Get then
                 toClient.outrecord(M.Get)
              else
                 toClient.outrecord("?");
           end - InputEvent
        end- inspect -;
     end - while -;
     DisableInputEventsUNIX(toClient.GetChannelNo);
     toClient.Close;
  end -- ClerkProcess --;
  ! ------------------------------------------------ file: operatorprocess.sim;
  External class IOProcess, ServerSwitch, ServerSocket,
        ClerkProcess,Memory;
  IOProcess class OperatorProcess(M); ref(Memory) M;
  begin
     ref(ServerSwitch) SwitchBoard;
     ref(ServerSocket) toClient;
     ref(ClerkProcess) aClerk;
     ref(ProcessEvent) Event;
     ref(InternetAddress) Us;
     Us:-new InternetAddress; Us.SetPort(11475);Us.SetStream;
     SwitchBoard:-new ServerSwitch(Us);
     if not SwitchBoard.Open(4) then
        Error("Can not open Socket");
     EnableInputEventsUNIX(SwitchBoard.GetChannelNo);
     EnableCancelEvents;
     while not Event in CancelEvent do
     begin
        Event:-WaitEvent; ! -- OperatorProcess waits --;
        inspect Event ! -- OperatorProcess continues --;
        when InputEvent do
        begin
           toClient:-SwitchBoard.NextRequest(new ServerSocket);
           aClerk:-new ClerkProcess(toClient,M);
           PMG.RegisterProcess(aClerk);
           aClerk.Start;
        end - inspect -;
     end - while ;
     DisableInputEventsUNIX(SwitchBoard.GetChannelNo);
     DisableCancelEvents;
     SwitchBoard.Close;
  end -- OperatorProcess --;
```

```
! ------------------------------------------------ file: adminprocess.sim;
External class ProcessManager, IOProcess, BasicProcessEvent, Memory;
IOProcess class AdminProcess(PMG,M);
   ref(ProcessManager) PMG; ref(Memory) M;
begin
   ref(ProcessEvent) Event;
   EnableInputEvents(Sysin);
   while not sysin.endfile do
   begin
      Event:-WaitEvent; ! -- AdminProcess waits --;
      inspect Event ! -- AdminProcess continues --;
      when InputEvent do
      begin
         inimage;
         if not Sysin.Endfile then
            outtext("Stored value: "& M.get); outimage;
      end;
   end;
   DisableInputEvents(Sysin);
   PMG.CancelAll;
end -- AdminProcess --;
! ------------------------------------------------ file: multipleserver.sim;
begin ! Main program MultipleServer ;
External class ProcessManager,
      OperatorProcess, Memory, AdminProcess;

   ref(ProcessManager) PMG;
   ref(OperatorProcess) Operator;
   ref(AdminProcess) Watchdog;
   ref(Memory) M;

   PMG:- new ProcessManager;
   M:-new Memory(PMG);

   Operator:-new OperatorProcess(M);
   PMG.RegisterProcess(Operator);
   Operator.Start;
   Watchdog:-new AdminProcess(PMG,M);
   PMG.RegisterProcess(Watchdog);
   Watchdog.Start;
   PMG.Run; ! -- Main program waits --;
            ! -- Main program continues --;
end
compilation:
   simmake -c singleserver simpleclient mutipleserver -l
linking:
   simld singleserver -l -d
   simld simpleclient -l -d
   simld mulitpleserver memory operatorprocess clerkprocess \
      adminprocess -l -d
```

Start a server and then a client in a process (shell window) each. Executing them with the debugger and 'trace on' gives a good feeling for how the processes in the Server program and the Clien-Server Unix processes interact.

6 Detailed Interfaces

6.1 InternetAddress

class InternetAddress;

Machines on InterNet are really named by 32-bit integers, call one of: SetHostbyNumber, SetHostbyInetNumber OR SetHostbyName to define the address, and you can then call:
HostName - to get the Machine name in the form "biur.dna.lth.se" AND InetNumber - to get the real 32-bit Inet address
"Clients" usually use this class to convert from a machine Name to its InterNet address as a number.
"Servers" use this class to convert an InterNet address of a connected Client to the name of the Machine for authorization etc.
Initially an InterNetAddress object will refer to the same machine on which the program itself is executing. This will also be the effect after failed calls to SetHostbyName.

Port numbers are integers defined by the Server program. The simplest way is that both Server and Client knows about the number.
Call 'SetPort' to tell which Port number you will define services on ("Server"), or want to talk to ("Client").

This implementation only support the Protocol Stream/TCP, in the 'AF_INET' domain. Call 'SetStream' to use that protocol.

Super: none
Kind: instantiatable
Init: Define Port and Protocol (and also Host for Clients)
- SetHostbyName, SetHostbyNumber, or SetHostbyInetNumber
- SetPort and
- SetStream
Sequence: (
SetHostbyName/SetHostbyNumber/SetHostbyInetNumber
SetPort SetStream/SetProtocol
(HostName/InterNetNumber/
GetPort/GetDomain/GetType/GetProtocol))*

Initialization operations

SetHostbyName

boolean procedure SetHostbyName(HostName);
text HostName; ! 'name' of machine to talk to;

Give the address of the other machine by its 'name' in the form <machine>.<site>. etc, example: Biur.dna.lth.se. Returns true if the machine was identified and the name thus was translated to a internet number. Returns false otherwise and the effect is that the machine denoted by this InterNetAddress object is the machine on which it is executing (which is also true initially).

SetHostbyNumber

procedure SetHostbyNumber(n); integer n;

Set internet directly as a 32-bit integer. There are also three sepcial cases:
INADDR_ANY - used by Servers to specify acceptance of clinets on any machine.
INADDR_LOOPBACK - Debugging facility, send back to the same socket.
INADDR_BROADCAST - Send to many machines, note this value has to be masked, not to transmit to the whole world.

SetHostbyInetNumber
procedure SetHostbyInetNumber(i1,i2,i3,i4); integer i1,i2,i3,i4;
Set hostnumber by integers grouped in four 8-bit groups, such as 130.21.16.110.

SetDGram
procedure SetDGram;
Define this connection to use DataGram/UDP protocol.

SetStream
procedure SetStream;
Define this connection to use Stream/TCP protocol.

SetProtocol
procedure SetProtocol(Master);
ref(InterNetAddress) Master;
Define this connection to use the same Protocol as Master does.

SetPort
procedure SetPort(Port);
integer Port;
Define this connection to use the port-number 'Port'.

Enqueries operations

InterNetNumber
integer procedure InterNetNumber;
Return the Internet number of the 'other' machine, as set directly or looked up through the used hostname. Returns Zero if not initiated, or the machine not identified.

HostName
text procedure HostName;
Return the name of the other machine in textual form. Retruns Notext if none of the Set-routines has been called, or the machine could not be identified.

GetPort
integer procedure GetPort;
Return the connection port-number defined for this InternetAddress.

GetDomain
integer procedure GetDomain;
Return the Domain Family of the connection. Currently allways returns 'AF_INET'

GetType
integer procedure GetType;
Return the 'Type' of the connection, determined by Protocol chosen. Currently allways returns 'SOCK_STREAM'

GetProtocol
integer procedure GetProtocol;
Return the Protocol chosen for the connection.
Currently allways returns 'IPPROTO_TCP'

6.2 SocketBasics

class SocketBasics;
This class contains a collection of c-routine interfaces useful when implementing Client/Server communication using Sockets. This class is abstract, its rou-

tines are intended to be used by subclasses only, they are therefor declared protected.
Super: none.;
Kind: Abstract.;
Subclasses: SocketIO ClientSocket/ServerSocket
SocketSwitch;
Init: see subclasses.;
Sequencing: (IsOpen / Open (GetChannelNo/<subclass ops>)* Close)*;
Intentions: Subclasses must implement an Open procedure.;

Operations

Close

Boolean procedure Close;

Close - close the channel represented by this Socket-object. Returns True on success, false on failure.

GetChannelNo

integer procedure GetChannelNo;

Return the UNIX channel number of an open Socket

IsOpen

Boolean procedure IsOpen;

Return True if this Socket is currently open

6.3 SocketIO

SocketBasics class SocketIO;

The Read/Write operations are directly calling the I/O operations provided for Sockets. They are thus not supporting any structure on the communication, but this has to be implemented by the application. One Write can be matched by many Reads (using smaller buffers) or several Writes can be matched by one Read (e.g. if the reader has been delayed). The Read/Write operations can be used to write Simula programs communicating with existing servers or clients.
In many cases the communication is structured as an exchange of messages. OutRecord/InRecord supports this organization. One call to OutRecord on the sending side always match one call of InRecord on the receiving side. These operations use a specific format of the messages and are only useful if there are Simula programs in both ends (or if a program written in another language can be adapted to use the same format).
A call to Read or InRecord will hang if there are no data available. The functionality provided by the library SimIOProcess can be used to avoid calling these operations until data is actually present (and thus avoiding the possible hang). See this library for details.
SocketIO is an abstract class with two concrete subclasses: ClientSocket and ServerSocket.
Super: SocketBasics;
Kind: Abstract;
Subclasses: ClientSocket and ServerSocket;
Init: see subclasses;
Sequencing: see subclasses;

Operations

Write

integer procedure Write(T);
text T; ! the bytes to send;

Write the content of T to the Socket. Return number of bytes actually written. Returns <0 it there is a problem, like the program in the other end did close its Socket.

Read
integer procedure Read(T);
text T; ! Text to fill, its length is max to read;

Attempt to fill T with info info read from the Socket. Waits unitl info available. Returns, N, number of bytes read and stored in T.sub(1,N). Returns <=0 if there is a problem, like the program in the other end did close its Socket. The OS often enforce a maximum size of messages that can be returned in one go, like 4096bytes under Unix. In such cases long messages must be read by several calls to Read.

OutRecord
boolean procedure OutRecord(T);
text T; ! the info to send;

Send a sequence of bytes of information to the other end. The info in T might be a readable ASCII sequence or some binary information. Returns True if all info send OK.

InRecord
text procedure InRecord(T);
text T; ! Suggested buffer for received message.;

Wait for and read the next message from the Socket. T is used for the message if long enough, if not a new buffer is allocated. The result of InRecord is either the new buffer or a subtext of T containing the received information.

Endfile
Boolean procedure Endfile;

Return true if last I/O operation did not input/output anything. This happens if the program in other end closed his socket (or died), but also in other situations.

6.4 ClientSocket

SocketIO class ClientSocket(Address);
ref(InterNetAddress) Address; ! Info on Server to talk to.;

Class to use by a Client for communicating over a Socket. ClientSocket objects has an object of class InternetAddress as a parameter rather than a textual file-name. The `Open' operation is implementing the elaborate sequence of operations needed to establish a connection with a a Server. Using the TCP/Stream protocol this involves exchange of several messages and a time-out in case the external Server does not respond. The time to perform an Open operation is thus unpredictable. Operations for I/O are those described in SocketIO.
Super: SocketBasics, SocketIO
Kind: Concrete.
Init: the parameter must denote a valid InternetAddress object.
Sequencing: (IsOpen / Open (GetChannelNo/ <IO-ops>)* Close)*;
: <IO-ops>=(OutRecord / InRecord / EndFile)* /
: (Read / Write / EndFile)*

Operations

Open
boolean procedure Open;

Open a Socket communication link to the Server. Wait until the Server responds, then return true. If the Server does not respond (or maybe does not exist) Open will return False after some timeout period. The parameters to the object, ServerInet, ServerPort are used to address the Server, and a program executing on the Server to talk to.

6.5 ServerSocket

SocketIO class ServerSocket;

Object used by Server to acually talk to a Client. The parameter, Channel, is normally the result returned by a call to SocketService.NextRequest. ServerSocket object is also similar to a Simula file object. The address of the ClientSocket object it will be connected to is, however, not known beforehand since the Client will take the initiative to call the Server. The `Open' operation is usually not called directly by the application program but is called by the NextRequest operation in SocketService. As a result the address of the Server-Socket is defined (as an InternetAddress object denoting the connected Client machine). Also notice that if Sockets are not propperly closed, say if the program is terminated due to some error, it takes a while (some minutes) for Unix to realize that processes using the Socket are gone. In the meantime it is not possible to start a new Server using the same Port number. Operations for I/O are those described in SocketIO.

Super: SocketConnection, SocketBasics.
Kind: Concrete.
Init: SocketSwitch.NextRequest calls Open.
Sequencing: ;
: (IsOpen / Open (GetChannelNo/GetClientAddress/<IO-ops>)* Close)*
: <IO-ops>=(OutRecord / InRecord / EndFile)* /
: (Read / Write / EndFile)*

Operations

Open

procedure Open(ChannelNumber,InternetAdr);
integer ChannelNumber; ! Unix Channel, from by 'Accept' via NextRequest;
ref(InterNetAddress) InternetAdr; ! Address of Client that connected to us;

Initiate this object as open as a result of an succesfull call to Accept. This is managed by NextRequest in class SocketSwitch, so there is no need to call this operation directly.

GetClientAddress

ref(InterNetAddress) procedure GetClientAddress;

Return the Internet address of the Client this Socket is connected to.

6.6 ServerSwitch

SocketBasics class ServerSwitch(ServerAddress);
ref(InterNetAddress) ServerAddress; !- Defines port and protocol.;

There is usually only one object of this class in a Server where it defines a Port with a Protocol. External requests to connect are handled by the operation NextRequest. It initializes an object of class ServerSocket to be used to service the connection.

The functionality provided by the library SimIOProcess can be used to avoid calling NextRequest until a request is actually present (and thus avoiding the possible hang). See this library for details.

Class used by a server to define a Server-Port. Receives requests from Clients (see class ClientConnection) who whants to talk to us. Our result of the request is another channel which is then used for the actual talking (see ServerConnection).

Super: SocketBasics.;
Kind: Concrete. ;
Init: the parameter must be a valid, unique, Port number. ;
Sequencing: (IsOpen/Open (GetChannelNo/<IO-ops>)* Close)*;
<IO-ops>=NextRequest ;

Operations

Open

```
boolean procedure Open(Qlength);
integer Qlength; ! Maximum number of unanswered Clients waiting ;
```

Open the socket-port for incomming requests. Qlength is the maximum number of unanswered Clients we allow. If more Clients have called tried to contact us (ClientSocket.Open) but are not yet answered we are considered overrun and further calls are refused.

NextRequest

```
ref(ServerSocket) procedure NextRequest(ReceiverSocket);
ref(ServerSocket) ReceiverSocket; ! Will be connected to Client;
```

Wait until there is a Client trying to contact us (calling ClientSocket.Open). Then return with ReceiverSocket initiated to use the channel to the Client. ReceiverSocket also carries the InternetNumber of the contacting Client. When this rendezvous has taken place, both Server and Client continue their execution. Returns None if the there was some error situation detected.

7 Index to classes and procedures

6.1 InternetAddress, 13

SetHostbyName, 13
SetHostbyNumber, 13
SetHostbyInetNumber, 14
SetDGram, 14
SetStream, 14
SetProtocol, 14
SetPort, 14
InterNetNumber, 14
HostName, 14
GetPort, 14
GetDomain, 14
GetType, 14
GetProtocol, 14

6.2 SocketBasics, 14

Close, 15
GetChannelNo, 15
IsOpen, 15

6.3 SocketIO, 15

Write, 15
Read, 16
OutRecord, 16
InRecord, 16
Endfile, 16

6.4 ClientSocket, 16

Open, 16

6.5 ServerSocket, 17

Open, 17
GetClientAddress, 17

6.6 ServerSwitch, 17

Open, 18
NextRequest, 18