Interner Bericht

VISIT - a Visualization Interface Toolkit

Version 1.0

Thomas Eickermann, Wolfgang Frings

FZJ-ZAM-IB-2000-16

Dezember 2000

# Contents

1 Introduction .................................................. 3  
  1.1 Credits and Copyrights .................................... 4  

2 General Concepts of Visit .................................... 5  
  2.1 Conventions ............................................... 5  
  2.2 Client – Server Connections ............................... 6  
  2.3 Visit Requests ............................................ 6  
  2.4 Message envelope ......................................... 6  
  2.5 Message data .............................................. 7  
  2.6 Visit and Perl5 ........................................... 8  

3 The Visit Client API .......................................... 9  
  3.1 Usage ........................................................ 9  
  3.2 visit_connect ............................................ 10  
  3.3 visit_connect_to_host .................................... 11  
  3.4 visit_connect_to_file .................................... 12  
  3.5 visit_disconnect ......................................... 13  
  3.6 visit_configure .......................................... 14  
  3.7 visit_send_4d ............................................ 15  
  3.8 visit_send_4d_os ......................................... 17  
  3.9 visit_send_string ........................................ 18  
  3.10 visit_recv_4d ............................................ 19  
  3.11 visit_recv_4d_os ........................................ 21  
  3.12 visit_recv_string ....................................... 22  

4 The Visit server API ......................................... 23  
  4.1 Usage ........................................................ 24  
  4.2 visit_svr_init_socket .................................... 24  
  4.3 visit_svr_init_socket_raw ............................... 26  
  4.4 visit_svr_init_file ....................................... 27  
  4.5 visit_svr_connect ....................................... 27  
  4.6 visit_svr_disconnect .................................... 28  
  4.7 visit_svr_shutdown ....................................... 28  
  4.8 visit_svr_getLid ......................................... 29  
  4.9 visit_svr_get_request ................................... 30  
  4.10 visit_svr_read_data ................................... 30  
  4.11 visit_svr_write_data ................................... 31  
  4.12 visit_svr_ack2 ......................................... 32  
  4.13 visit_svr_configure .................................... 32  
  4.14 visit_svr_socket_lsd, visit_svr_socket_csd, .......... 33
## 5 The AVS/Express *visit*–server

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 visitserver</td>
<td>35</td>
</tr>
<tr>
<td>5.2 visitreader</td>
<td>37</td>
</tr>
<tr>
<td>5.3 visitwriter</td>
<td>39</td>
</tr>
<tr>
<td>5.4 multiplexer</td>
<td>41</td>
</tr>
<tr>
<td>5.5 VisitServer</td>
<td>42</td>
</tr>
<tr>
<td>5.6 VisitReader</td>
<td>44</td>
</tr>
<tr>
<td>5.7 VisitWriter</td>
<td>45</td>
</tr>
<tr>
<td>5.8 Multiplexer</td>
<td>48</td>
</tr>
</tbody>
</table>

## 6 *seap* — the *service announcement* protocol

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 The <em>seap</em>–server</td>
<td>50</td>
</tr>
<tr>
<td>6.2 The <em>seap</em> client functions</td>
<td>50</td>
</tr>
<tr>
<td>6.2.1 Usage</td>
<td>50</td>
</tr>
<tr>
<td>6.2.2 <em>seap</em>–publish</td>
<td>51</td>
</tr>
<tr>
<td>6.2.3 <em>seap</em>–unpublish</td>
<td>51</td>
</tr>
<tr>
<td>6.2.4 <em>seap</em>–query</td>
<td>51</td>
</tr>
<tr>
<td>6.3 <em>seap</em> demo clients</td>
<td>52</td>
</tr>
</tbody>
</table>

## 7 Tools

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 <em>seap</em> – monitoring the <em>seap</em>–server</td>
<td>53</td>
</tr>
<tr>
<td>7.2 <em>vbroker</em> – attaching multiple visualizations</td>
<td>54</td>
</tr>
<tr>
<td>7.2.1 The ’Client connection / Simulation’ panel</td>
<td>54</td>
</tr>
<tr>
<td>7.2.2 The ’Server connections / Visualizations’ panel</td>
<td>55</td>
</tr>
<tr>
<td>7.2.3 Example session</td>
<td>55</td>
</tr>
</tbody>
</table>

## 8 Demo Programs

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1 Test clients and servers</td>
<td>57</td>
</tr>
<tr>
<td>8.1.1 vclient.c</td>
<td>57</td>
</tr>
<tr>
<td>8.1.2 vserv.c</td>
<td>57</td>
</tr>
<tr>
<td>8.1.3 vclient.pl</td>
<td>58</td>
</tr>
<tr>
<td>8.1.4 vserv.pl, tserv.pl</td>
<td>58</td>
</tr>
<tr>
<td>8.1.5 VisitSimpleEg (AVS/Express)</td>
<td>58</td>
</tr>
<tr>
<td>8.1.6 fvclient.f</td>
<td>58</td>
</tr>
<tr>
<td>8.1.7 sclient.c, querytime.c</td>
<td>58</td>
</tr>
<tr>
<td>8.2 Game of Life</td>
<td>59</td>
</tr>
<tr>
<td>8.2.1 VisitGoLEg (AVS/Express)</td>
<td>59</td>
</tr>
<tr>
<td>8.2.2 tkgol.pl</td>
<td>62</td>
</tr>
</tbody>
</table>

## 9 Installation and Porting

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 Prerequisites</td>
<td>63</td>
</tr>
<tr>
<td>9.2 Quick Installation</td>
<td>63</td>
</tr>
<tr>
<td>9.3 Test the installation</td>
<td>64</td>
</tr>
<tr>
<td>9.4 Configure options</td>
<td>65</td>
</tr>
<tr>
<td>9.5 Porting hints</td>
<td>67</td>
</tr>
<tr>
<td>9.5.1 Fortran issues</td>
<td>67</td>
</tr>
<tr>
<td>9.5.2 Data types on new platforms</td>
<td>67</td>
</tr>
<tr>
<td>9.5.3 Defining new data types</td>
<td>67</td>
</tr>
</tbody>
</table>
# List of Figures

4.1 schematic diagram of a simple *visit*-server. ................................. 24

5.1 AVS/Express network which uses the *visit* macros and the Panel of the *visitserver* macro. ............................................................... 34

7.1 seap displaying a couple of services related to the Game of life demo. ........ 53
7.2 VBroker client and server panels in a typical Game of Life session as described in the example section 7.2.3. ................................. 54

8.1 Visualization and steering for the Game of Life simulation with AVS/Express. . . 60
8.2 Top-level network for this example. ................................................ 61
8.3 Network for the communication between AVS/Express and cgol. ................... 61
8.4 Visualize and steer the Game of Life simulation with Perl/Tk ...................... 62
Visit - Visualization Interface Toolkit
Seap - Service Announcement Protocol

Copyright (C) 2000, Forschungszentrum Juelich GmbH, Federal Republic of Germany. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Any publications that result from the use of this software shall reasonably refer to the Research Centre’s development.
- All advertising materials mentioning features or use of this software must display the following acknowledgment:
  This product includes software developed by Forschungszentrum Juelich GmbH, Federal Republic of Germany.
- Forschungszentrum Juelich GmbH is not obligated to provide the user with any support, consulting, training or assistance of any kind with regard to the use, operation and performance of this software or to provide the user with any updates, revisions or new versions.

THIS SOFTWARE IS PROVIDED BY FORSCHUNGSZENTRUM JUELICH GMBH "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL FORSCHUNGSZENTRUM JUELICH GMBH BE LIABLE FOR ANY SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE ACCESS, USE OR PERFORMANCE OF THIS SOFTWARE.
Chapter 1

Introduction

With the increasing capabilities of both supercomputers and graphical workstations new modes of operation become feasible for numerical simulations that are traditionally performed in batch processing. Connecting a workstation to a compute–server allows for interactive monitoring and control of such simulations. Buzz-words in that field are online-visualization, interactive simulations, or computational steering. Of course, there is a need for programming tools that support the development of interactive applications. A couple of such tools is freely available from various institutions (CUMULVS, CSE, SCIRun, OViD). But when — during the "Gigabit Testbed West" project, a testbed for the new German Gigabit Science Network, the G-WiN — the need for such a tool arose, we found them being either to simplistic or to complex and decided to develop our own tools, named visit (visualization interface toolkit). What we needed is basically a set of functions that allow for establishing a connection between simulation and visualization, exchanging data and eventually shutting down the connection again. Basic design considerations were:

- The simulation is considered 'more important' than the visualization (CPU–time on supercomputers is expensive) and therefore should be disturbed as little as possible by failures or slow operation of the visualization. This means that all operations (like sending data to be visualized or receiving new parameters) have to be initiated by the simulation and are guaranteed to complete (or fail) after a user–specified timeout. The visualization acts as a server that dispatches the simulations requests.

- It should be possible to transfer at least simple datatypes like strings, integers and floats and arrays of integers and floats. The implementation should not inhibit later extensions.

- Data conversions should be performed transparently for the user. However, since most modern architectures use IEEE format, this is restricted to byte-order conversions (to avoid expensive conversions into external data representations like XDR).

- The simulation should be able to connect to the visualization using "service–names" rather than host–name and port–number. This on the one hand allows to avoid port–number conflicts and on the other hand enables the user to start the visualization on any workstation (or even on different workstations, one at a time). However, this requires some kind of naming–service.

- The data–transport should use IP, the only protocol that is available on all platforms, but the API should allow for other mechanisms that we may implement in the future (e.g. MPI, using the MPI-2 process attachment or low–level network protocols like Myrinet).

- It should be possible for a simulation/visualization to connect to more than one partner at a time.

- For the simulation, language bindings for at least C and FORTRAN are a must.

- In the Gigabit Testbed applications, mainly AVS/Express was used for visualization. The complete visit–functionality should be available through AVS/Express modules so that there is no need for the Express programmer to write extra C–code to access them.
• Both the API and the implementation should be lean, to simplify usage and porting to new platforms.
• Security was considered a minor issue for visit. Authorization of simulation and visualization is based on a user-specified password. All data (including the password) is transmitted without any encryption. This will be changed in a later version.

Bundled with visit is a simple name-service named seap (service announcement protocol). A visualization can register its service(s) at a seap-server. The simulation can query this information (consisting of a host-name and a port-number) and use it to connect to the visualization. Currently, the seap-server is implemented in perl and has no access-restrictions. Anyone can register services. Anyone who knows the passwd associated with a service can query and unregister this service.

The current implementation of visit uses TCP/IP sockets for the connection between simulation and visualization. Besides that, it is also possible for the simulation to write the data to a file and for the visualization to read the data from such files, using the same send/receive calls as for a socket-connection. This is intended to be used for offline-visualization, where the simulation data is recorded in advance. For the simulation, the API consists only of a few function calls, making it very easy to use visit in a simulation. Language bindings are available for C, FORTRAN, and Perl. For the visualization-side, visit has language bindings for C and Perl. The distribution contains demo clients and servers in all supported languages. A complete server for AVS/Express is also included.

The rest of this document is organized as follows. The next sections describe the visit API for the simulation (client) and visualization (server) side for all language bindings. One section is dedicated to seap. After that, the demo programs included with the source distribution are discussed. The last part contains installation instructions and hints for porting.

1.1 Credits and Copyrights

visit uses minilzo, which is part of the LZO-library by Markus Franz Xaver Johannes Oberhumer. LZO is free software that is distributed under the terms of the GNU General Public License (either version 2 or any later version).

visit itself is copyrighted software of the Research Centre Jülich GmbH. It can be downloaded from the Research Centres web-server:

http://www.fz-juelich.de/zam/Angebote/Angebote-e.html

It can be used and redistributed under the conditions stated in the DISCLAIMER file that is contained in the distribution. The AVS/Express modules are also available at the IAC.
Chapter 2

General Concepts of Visit

visit is a library for point-to-point communication between two independent applications (like a simulation and a visualization) using a client–server model. Building a visit–client is as simple as file I/O using open, read, write, and close. To implement a visit–server is slightly more complex, but supported by a 'server–toolkit' that is part of the library. The visit API abstracts from the underlying network protocol, so the same functions can be used for different types of communication.

2.1 Conventions

Due to the client–server architecture of visit we will use the terms visit–client and visit–server (or just client and server) for the simulation and visualization parts of a coupled application using visit. This also reflects the fact that the usage of visit is not limited to steering and interactive simulations. In the description of the API we use the following typesetting rules. For each function, the C function declaration, the FORTRAN subroutine header and the Perl–Call are given. Parameters with equal names have identical meanings in all language bindings. Following this declaration, there is a short description for each parameter. visit is implemented in C. Our goals to build the FORTRAN bindings as a thin layer on top of the C implementation, to be compatible with FORTRAN 90, and to keep it portable, lead to some compromises.

C functions which return a status will return 1 in case of success and 0 otherwise. Functions that return a non-negative integer (like a connection descriptor), will return -1 in case of an error. Functions that return a pointer will return NULL in case of an error.

Constants are defined as macros in the header–file visit.h

FORTRAN bindings use subroutines not functions. For all functions that are non–void in C, the subroutine returns a status (0 or 1) in the last parameter IERROR. If the C–function returns more than just a status, the subroutine returns that value in its first parameter.

Constants are defined as parameters in the header–file fvisit.h

Perl bindings are more Perl–stylish object–oriented. Parameters are generally named like e.g. in the Perl-Tk modules. This allows for default arguments and some function overloading. In the description, optional arguments are placed in brackets ([ ]). The default values are put in parenthesis. If a C–function returns a status, the corresponding Perl–function will also return 0 or 1. For C-functions that return more than a status (and -1 or NULL in case of an error) the Perl–function will return ‘undef’ in case of an error. For both C and FORTRAN, connections and other ‘objects’ are accessed via integer descriptors. In Perl, object references are used instead.

The message data can either be stored in standard Perl arrays which is not very efficient for large amounts of data (mainly in terms of memory usage) or in a packed binary form.
However, handling arrays with a few thousands of elements is no problem. Instead of the integer constants in C and FORTRAN, strings are used (e.g. ‘INT32’ instead of VISIT_INT32 (=2)).

### 2.2 Client – Server Connections

The communication between visit–server and –client is connection–oriented. This means that client and server must explicitly establish a connection before they can exchange data. In detail, the visit–server will perform all the initializations that are required to be able to accept clients. After that he will register his service at the seap–server (the information a client needs to find the server is associated with a service–name and a key). A visit–client queries the seap–server for this data (by specifying service–name and key) and connects to the visit–server. During the startup, client and server negotiate certain parameters of the connection (like byte order and authorization).

E.g. in the TCP/IP case, the visit–server will open a listening socket and then register its hostname and portnumber at the seap–server. The visit–client queries these parameters from the seap–server by specifying the service–name and key. The client the connects to the visit–server and authorizes himself by sending the key.

Once established, a connection stays open, until it is either explicitly shut down by one of the partners, a partner dies or exits. Currently, visit does not contain sophisticated error–recovery mechanisms. When one of the partners detects an error during communication (timeouts or invalid response from the other partner) he shuts the connection down.

### 2.3 Visit Requests

After the connection is established, client and server can exchange messages. Such a message consists of an envelope and the actual data. The envelope contains meta–information like message size and type (see below for details). It is always the client who initiates a message transfer by sending a request to the server, no matter in which direction the message will flow. The first action of the client is to check whether the server is ready to accept a request. (It is a property of the connection, how long the client will wait for the server to respond until he assumes that the server is not ready.) If the server is ready, the client proceeds, otherwise the request is canceled.

In case of a send request, the client then sends a message id, the message envelope, and the data. In case of a receive request, he sends just the id and an envelope. The server responds by sending back a message consisting of envelope and data. It is not guaranteed that this message matches the requested one. However, to avoid buffer overflows, the client rejects messages that are larger than requested.

### 2.4 Message envelope

Currently, a message envelope contains the following data. Due to the limitation to at most 4D–arrays, the size of the envelope is constant. Depending on the actual dimension, not all entries may be significant.
2.5 MESSAGE DATA

timestamp a float value (no specific meaning).
vtype an integer value that specifies the type of the message data. This value is used to calculate the size of the message and to perform necessary format conversions between client and server (see below for a list of supported datatypes).
ndim the dimensionality of the message data. The dimension may be 1, 2, 3, or 4 (1 for a scalar).
n1 array extension in the first dimension (1 for a scalar)
n2 array extension in the second dimension (not used, if ndim<2)
n3 array extension in the third dimension (not used, if ndim<3)
n4 array extension in the forth dimension (not used, if ndim<4)
o1 offset in first dimension (no specific meaning)
o2 offset in second dimension (no specific meaning)
o3 offset in third dimension (no specific meaning)
o4 offset in forth dimension (no specific meaning)
s1 stride in first dimension (no specific meaning)
s2 stride in second dimension (no specific meaning)
s3 stride in third dimension (no specific meaning)
s4 stride in forth dimension (no specific meaning)

For correct message delivery, the parameters vtype, ndim, n1, ... n4 are significant. The other parameters are for convenience only. The timestamp is intended to identify a data–set in a series of similar data, as it often occurs in simulations. offset and stride are intended to indicate that the transmitted data is only part of a larger field (starting at an offset, and being subsampled with a stride). However, these parameters can be used for other purposes or be ignored completely.

2.5 Message data

Currently, visit is able to transfer strings and arrays of integer (1, 2, or 4 bytes) and floating point (8 byte) numbers. An array may be no more than 4–dimensional. Scalar values are treated as a 1–dimensional arrays of length 1. The reason for the limitation in the dimension is to make the API simpler — and not laziness as you may suspect (this affects mainly the AVS/Express modules). We chose 4 as a limit because in visualization one usually deals with at most 3D–data. The forth dimension can e.g. be used to transfer 3D–vector–fields or time–series of 3D–data.

Currently the following datatypes are supported by visit:

<table>
<thead>
<tr>
<th>datatype</th>
<th>C, FORTRAN name</th>
<th>Perl name</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit floats (typically double)</td>
<td>VISIT_FLOAT64</td>
<td>'FLOAT64'</td>
</tr>
<tr>
<td>32-bit integers (typically int or long)</td>
<td>VISIT_INT32</td>
<td>'INT32'</td>
</tr>
<tr>
<td>16-bit integers (typically short)</td>
<td>VISIT_INT16</td>
<td>'INT16'</td>
</tr>
<tr>
<td>8-bit integers (typically unsigned char)</td>
<td>VISIT_BYTE</td>
<td>'BYTE'</td>
</tr>
<tr>
<td>0-terminated C-strings</td>
<td>VISIT_STRING</td>
<td>'STRING'</td>
</tr>
</tbody>
</table>

The values in the second and third column are the values/names under which the types have to be specified in the vtype–parameter of the envelope. Not all datatypes may be available on all platforms (e.g. T3E is lacking INT16).

visit provides automatic byte order conversions between client and server if necessary. Currently, this is the only data conversion between client and server. It is always performed by the server following the idea that the extra load on the client (the simulation) should be as low as possible. This means that in contrast to e.g. MPI, the data must have the same representation (except for the byte order) on both sides of the application.

It is in the users responsibility to specify correct data types. E.g. on a Cray T3E the C–type
short matches VISIT_INT32 on many other it is VISIT_INT16. Currently, there is only little help: we have typedef’d the data–types vint16 and vint32 in the visit–header file to the proper integer type — where available. Strings may be sent using ndim=1 and n1=strlen(data) in C programs. However, due to non–portable string representations in FORTRAN, there is an additional function visit_send_string (see 3.9) to send strings. This function may also be used in C programs. There is no way to transfer arrays of strings with a single function call.

In C and FORTRAN it is generally assumed that the data is contained in a continuous piece of memory. n1 is the ‘fastest index’, therefore a 4D–array to be handled by visit would be a[n4][n3][n2][n1] in C (and Perl) and A(N1,N2,N3,N4) in FORTRAN. The data–parameter of the visit functions has to be the address of the first data item.

2.6 Visit and Perl5

The Perl interface to visit is different from the C and FORTRAN bindings. We tried to keep it closer to what is common practice for many OO–style Perl modules. As already mentioned, a visit–connection is represented by a Visit–object in Perl, not by an integer descriptor like in C and FORTRAN. Therefore, most of the visit–functions are object–methods in Perl. Generally, all parameters to those methods are ‘named’ (like -timestamp => 1.1) instead of ‘positional’. With respect to the envelope, there are two other differences. The first is related to the dimension, offset, and stride. Instead of 'unrolling’ these parameters in the parameter list of the methods, the Perl interface uses references to arrays containing those values, e.g. -dim => [ $n1, $n2, ... ] instead of -n1 => $n1, -n2 => $n2, ...

There is no ndim parameter. Instead, the dimensionality of the data is the length of the -dim–array.

The second difference concerns the receive functions, where envelope information has to be returned to the caller. In C this is done by passing the these parameters per reference. In Perl, you may add var to the name of a parameter and pass a reference to the variable containing the data. To obtain the timestamp, you would pass -timestampvar => $timestamp to a method that receives a message. If you know what you are doing, you may choose to ignore the timestamp that is part of the received message and use the standard form -timestamp => $timestamp.

Note that when the parameter is already a reference, the use of this mechanism does not take a 2nd reference. It only indicates, that the envelope information should be passed back to the parameter. If you use -dimvar => @dim, the dimensions will be returned in @dim, if you use -dim => @dim they will not. In the description of the API those parameters that may obtain a var postfix are noted with: (&).

Another thing specific to perl is how the data is passed. In C and FORTRAN, you pass the address of the first element of a continuous block of memory containing all the data. In Perl, you pass a reference to a normal 1...4–D Perl array. For the size of the array, the -dim parameter is determining. If the array is too small or contains undef values, the missing values are transmitted as zeros. You may pass an optional parameter -flat. If set to a true value, the data parameter has to be a reference to a 1D–array of the proper size. The same holds for the methods that receive data. A reference to a 1...4–D array is returned. With -flat set to true, a reference to a 1D array is returned instead. If the -flat notation is used, the order of the data is like in C and FORTRAN: the first element of the -dim–array is the 'fastest index'.

You may also pass an optional parameter -pack (that is mutual exclusive to -flat). If set to a true value, the data will be packed into a Perl–string (for a receiving method) or has to be passed in a packed string (for sending methods). Since the data is stored in its natural representation, it uses far less space that an array of Perl scalars. The access to individual elements is more complicated, it requires unpack or pack/substr to read or modify individual elements of the data.
The API for the visit–client contains just five groups of functions. These groups are connect and disconnect for establishing and shutting down connections, send and receive for exchanging data and configure for modifying properties of a connection.

The first group establishes connections to the server. For each protocol that is supported by visit, there is at least one such function. Currently, there are functions for TCP/IP connections with or without using seap and for connections to files. The connect–functions are the only protocol–dependent functions. If successful, they return an integer connection descriptor (C, FORTRAN) or a connection object (Perl) that is used by all of the other functions to identify the connection (like a UNIX file descriptor).

3.1 Usage

To use the visit–client functions, put one of the following lines of code in your program:

```c
#include "visit.h" /* C */
```

```fortran
include 'fvisit.h' // FORTRAN
```

```perl
use Visit; # Perl
```
3.2 visit_connect

```c
int visit_connect(char *service, char *passwd, int pollinterval,
                 int maxpoll,int msg_timeout, int conn_timeout);
```

```fortran
SUBROUTINE FVISIT_CONNECT(VCD, SERVICE, PASSWD, POLLINTERVAL,
                           MAXPOLL,MSG_TIMEOUT, CONN_TIMEOUT,
                           IERROR)

INTEGER*4   VCD, POLLINTERVAL, MAXPOLL, MSG_TIMEOUT
INTEGER*4   CONN_TIMEOUT, IERROR
CHARACTER(*) SERVICE, PASSWD

$vcd = Visit->new(  -service => $servicename,
                    -passwd => $passwd,
                    [ -pollinterval => $pollinterval, (2) ]
                    [ -maxpoll => $maxpoll, (2) ]
                    [ -msg_timeout => $msg_timeout, (2) ]
                    [ -conn_timeout => $conn_timeout (-1) ]
)
```

Description:
The first thing a client has to do before it can send or receive data to or from a server is to establish a connection to that server. visit_connect uses seap to locate the server. To obtain the contact information from the seap–server, it needs both a service–name and a passwd. visit_connect tries at most maxpoll times (with a pause of pollinterval milliseconds inbetween) to query the seap–server. If the information is not available after that time, visit_connect returns without establishing the connection. If the client is not able to contact the server after it has obtained the contact–information from the seap–server, the connect will also fail. Currently, the connection is based on TCP/IP sockets, but in later versions other protocols that can also be registered using seap may be available. To use visit_connect you have to have a seap–server running somewhere at your site. Hostname (seap_server) and port number of this seap–server also have to be specified.

Two parameters that influence the general behavior of a visit–connection are msg_timeout and conn_timeout. Whenever the client starts a send or receive request, it will return if the server has not responded to that request after msg_timeout milliseconds. The assumption is that the server is busy and it does not make sense to block the client any longer. The connection remains open for later usage in that case. If a server has responded to a request, but the request could not be completed after conn_timeout milliseconds, it is assumed that something has gone bad at the server and the connection is shut down by the client. Both timeouts can be set to -1 to let the client wait forever. With conn_timeout=-1, the visit–connection will only be shut down when the socket dies. With these timeouts, the user can control how much delay from the server (visualization) the client (simulation) is willing to accept.

With the function visit_configure (3.6) both msg_timeout and conn_timeout can be modified.
### Parameters:

- **service**: a service-name that must have been published by the server.
- **passwd**: a string that is associated with the service. Unlike the service-name it cannot be queried from the seap-server.
- **pollinterval**: the seap-server is polled every 'pollinterval' milliseconds, until the 'service' is available.
- **maxpoll**: maximum number of polls before visit_connect is timed out.
- **msg_timeout**: the client waits for at most msg_timeout milliseconds after initiating a read or write request, before it assumes that the server is not ready for dispatching this request. If that happens, the request is canceled but the connection remains open.
- **conn_timeout**: when a request is not finished after conn_timeout milliseconds, the client assumes that the server is hanging and shuts down the connection.

### Return Values:

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>a non-negative connection descriptor</td>
<td>-1</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>a non-negative connection descriptor is stored in VCD, IERROR=1</td>
<td>VCD=-1 and IERROR=0</td>
</tr>
<tr>
<td>Perl</td>
<td>a Visit object</td>
<td>undef</td>
</tr>
</tbody>
</table>

### 3.3 visit_connect_to_host

```c
int visit_connect_to_host(char *host, int port, char *passwd,
                           int msg_timeout, int conn_timeout);
```

```fortran
SUBROUTINE FVISIT_CONNECT_TO_HOST(VCD, HOST, PORT, PASSWD,
                                   MSG_TIMEOUT, CONN_TIMEOUT, IERROR)
INTEGER*4 VCD, PORT, MSG_TIMEOUT, CONN_TIMEOUT, IERROR
CHARACTER(*) HOST, PASSWD
```

```perl
$vc = $visit->new( -host => $host,
                  -port => $port,
                  -passwd => $passwd,
                  [ -msg_timeout => $msg_timeout, (2) ]
                  [ -conn_timeout => $conn_timeout (-1) ]
);```

### Description:

This function is similar to visit_connect (see 3.2). The only difference is that it does not use seap to obtain the hostname and port number of the server. The user has to specify those parameters directly. The advantage is that no extra software (the seap-server) is required. A serious drawback (which lead us to build seap) is that ports tend to be 'in use' by other applications or just 'hang'. With seap, free ports can be choosen by the visit-server and used by the client without user interference. We strongly encourage you to use seap. Although visit_connect_to_host does not use seap a passwd has to be specified because this passwd is not only required for obtaining contact information from the seap-server but is also required for authorization at the server. See 3.2 for a more detailed description. Please note that Perl uses the function Visit->new for connections of all types. The type of the connection that is established depends on the parameters.
Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>hostname or IP–address of the server.</td>
</tr>
<tr>
<td>passwd</td>
<td>an authorization string that is associated with the service.</td>
</tr>
<tr>
<td>port</td>
<td>portnumber of the server on host.</td>
</tr>
<tr>
<td>msg_timeout</td>
<td>the client waits for at most msg_timeout milliseconds after initiating a read or write request, before it assumes that the server is not ready for dispatching this request. If that happens the request is canceled, but the connection remains open.</td>
</tr>
<tr>
<td>conn_timeout</td>
<td>when a request is not finished after conn_timeout milliseconds, the client assumes that the server is hanging and shuts down the connection.</td>
</tr>
</tbody>
</table>

Return Values:

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>a non–negative connection descriptor</td>
<td>-1</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>a non–negative connection descriptor is stored in VCD, IERROR=1</td>
<td>VCD=-1 and IERROR=0</td>
</tr>
<tr>
<td>Perl</td>
<td>a Visit object</td>
<td>undef</td>
</tr>
</tbody>
</table>

3.4 visit_connect_to_file

```c
int visit_connect_to_file(char *filename, char *mode, char *text);
```

```fortran
SUBROUTINE FVISIT_CONNECT_TO_FILE(VCD, FILENAME, MODE, TEXT,
                                   IERROR)
  INTEGER*4 VCD, IERROR
  CHARACTER*(*) FILENAME, MODE, TEXT

  $vcd = Visit->new( -filename => $filename, 
                    [ -mode => $mode, ('w') ]
                    [ -text => $text (undef) ]
  );
```

Description:

This function opens a pseudo–connection. All data that is send via this connection is stored in a file. The file is opened with mode mode which can be either "w" for writing or "a" appending. This parameter is directly passed to the C–function fopen, so the semantics are the same. Of course, receive–requests from a file–connection always fail. An optional text can be used to annotate the file. Please note that Perl uses the same function Visit->new for file and socket connections. The type of the connection that is established depends on the parameters.

When appending to a non–empty file, text is ignored. A limitation of the current implementation is that, when appending to a non–empty file, the writer and the original creator of the file must have the same byte order. visit_connect_to_file checks that condition and returns -1 if it is not fulfilled. This means that you cannot create a file on an Intel–Linux box and append to that file on a Sun Workstation.

Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>name of the file to write to.</td>
</tr>
<tr>
<td>mode</td>
<td>specifies, whether the file should be opened for writing or for appending.</td>
</tr>
<tr>
<td>text</td>
<td>an optional annotation for the file (NULL for no annotation).</td>
</tr>
</tbody>
</table>
3.5 visit_disconnect

```c
int visit_disconnect(int vcd);
```

```fortran
SUBROUTINE FVISIT_DISCONNECT(VCD, IERROR)

INTEGER VCD, IERROR

$vcd$->disconnect();
```

**Description:**
To close a connection to a server the client calls this function. The actual action depends on the type of the connection. For the TCP/IP connections, e.g. the sockets will be closed. For the file-connection, the file is closed. After calling `visit_disconnect`, the connection descriptor `vcd` is no longer valid. It may be reused by later calls of a `visit_connect` function.

**Parameters:**
- `vcd` a valid connection descriptor (or Visit object in Perl).

**Return Values:**
<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, Perl</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>IERROR=1</td>
<td>IERROR=0</td>
</tr>
</tbody>
</table>

Binding: C, FORTRAN, Perl

Return Values:

- **C**: a non–negative connection descriptor is stored in VCD, IERROR=1
- **FORTRAN**: VCD=-1 and IERROR=0
- **Perl**: a Visit object

Return Values:

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>a non–negative connection descriptor is stored in VCD, IERROR=1</td>
<td>-1</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>IERROR=1</td>
<td>IERROR=0</td>
</tr>
<tr>
<td>Perl</td>
<td>a Visit object</td>
<td>undef</td>
</tr>
</tbody>
</table>
3.6 visit_configure

int visit_configure(int vcd, int what, ...);

SUBROUTINE FVISIT_CONFIGURE(VCD, WHAT, VALUE, IERROR)

INTEGER VCD, WHAT, IERROR

$vcd->configure( -parameter_name => $new_value );

**Description:**
This function can be used to modify properties of an active connection. The parameters depend on the type of the connection. The parameter `what` is an integer that specifies which property shall be modified. Symbolic names for these integers are defined in the `visit.h` and `fvisit.h` header files. The next parameter is the new value of that property. Currently, the following properties can be modified:

**socket--connections:**

<table>
<thead>
<tr>
<th>what</th>
<th>property/parameter to change</th>
<th>type of parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>VISIT_MSG_TIMEOUT</td>
<td>msg_timeout</td>
<td>integer</td>
</tr>
<tr>
<td>VISIT_CONN_TIMEOUT</td>
<td>conn_timeout</td>
<td>integer</td>
</tr>
</tbody>
</table>

**file--connections:** none!

The Perl binding uses the names of the properties/parameters (as in the new method) to change instead of the integer `what`.

**Parameters:**

- `vcd` a valid connection descriptor (or Visit object in Perl).
- `what` an integer value, specifying the parameter to change.
- `...` the new value of that parameter.

**Return Values:**

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, Perl</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>IERROR=1</td>
<td>IERROR=0</td>
</tr>
</tbody>
</table>
3.7 visit_send_4d

int visit_send_4d(int vcd, int id, double timestamp,
                 void *data, visit_type vtype, int ndim,
                 int n1, int n2, int n3, int n4, ...);

SUBROUTINE FVISIT_SEND_4D(VCD,ID,TIMESTAMP,DATA,VTYPE,NDIM,
                N1,N2,N3,N4,IERROR)

INTEGER VCD,ID,VTYPE,NDIM,N1,N2,N3,N4,IERROR
REAL*8 TIMESTAMP

$ok = $vcd->send(  -id       => $id,
                -data     => $data,
                -vtype    => $vtype,
                [ -flat    => 1, (0) ]
                [ -pack    => 1, (0) ]
                [ -dim     => [ $n1, $n2, ... ],
                [ -offset  => [ $o1, $o2, ... ], (0, 0, ...) ]
                [ -stride  => [ $s1, $s2, ... ], (1, 1, ...) ]
                [ -timestamp => $timestamp, ]
)

Description:
This is the main function for sending data from the client to the server. It sends scalar values or
arrays of dimension ndim from 1 to 4. For each dimension, a size parameter (n1 ... n4) has
to be specified. The datatype of the data must be specified as a parameter (vtype).
In the C–bindings, the vtype can be arithmetically or’d with the type modifiers
VISIT_OFFSET and/or VISIT_STRIDE. In that case additional parameters (o1 ... o4 and/or
s1 ... s4) have to added to the parameter list. With the FORTRAN bindings you cannot use
these optional parameters. Instead, there is an additional function named visit_send4dos
which has fixed offset and stride parameters. See 2.5 and 2.4 for a detailed description of datatypes
and the 'envelope' parameters vtype, ndim, n1 .. n4, o1, .. o4, s1 .. s4, timestamp.

Parameters:
  vcd       a valid connection descriptor (or Visit object in Perl).
  id        an id that classifies the data. The server side can use this id to identify the
data (like the message tag in MPI).
  timestamp a parameter that is intended to further characterize the data, but has no
            specific meaning.
  data      pointer to the data to be send (even if a scalar is send).
  vtype     the datatype of the data (see list above).
  flat      indicates that data is a 1D array (Perl only).
  ndim      dimensionality of the data, must be 1,2,3, or 4.
  n1,n2,n3,n4 size of the data–array, only the first ndim values are used.
             In Perl, it’s a reference to an array with ndim entries.
  o1,o2,o3,o4 optional offsets into the data–array. (see n1..n4 for validity).
  s1,s2,s3,s4 optional strides of the data–array. (see n1..n4 for validity).
Return Values:

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Timed out with no data sent</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Success</td>
<td>Timed out with no data sent</td>
<td>Failure</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>IERROR=1</td>
<td>IERROR=0</td>
<td>IERROR=-1</td>
</tr>
<tr>
<td>Perl</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

Examples:

send a single 4-byte integer: vtype=VISIT_INT32,
    ndim=1, n1=1

3D double field a[nz][ny][nx]: vtype=VISIT_FLOAT64,
    ndim=3, n1=nx, n2=ny, n3=nz

The order of the field-dimensions is 'fastest index first', therefore in FORTRAN, the above parameters would apply to a field: A(NX, NY, NZ)
### 3.8. `visit_send_4d_os`

```c
int visit_send_4d_os(int vcd, int id, double timestamp,
    void *data, visit_type vtype, int ndim,
    int n1, int n2, int n3, int n4,
    int o1, int o2, int o3, int o4,
    int s1, int s2, int s3, int s4);
```

```fortran
SUBROUTINE FVISIT_SEND_4D_OS(VCD,ID,TIMESTAMP,DATA,VTYPE,NDIM,
    N1,N2,N3,N4,O1,O2,O3,O4,S1,S2,S3,S4,
    IERROR)

INTEGER*4 VCD,ID,VTYPE,NDIM,N1,N2,N3,N4,
INTEGER*4 O1,O2,O3,O4,S1,S2,S3,S4,IERROR
REAL*8 TIMESTAMP
```

**Description:**

This function has the same functionality as `visit_send_4d`. The only difference is that it has a fixed parameter list including offset and stride. See `visit_send_4d` (3.7) for a description of the parameters and return values. There is no Perl-binding for this function. The following calls are identical:

```c
visit_send_4d_os(vcd, id, timestamp, data,
    vtype, ndim,
    n1, n2, n3, n4,
    o1, o2, o3, o4,
    s1, s2, s3, s4);
```

```c
visit_send_4d(vcd, id, timestamp, data,
    vtype | VISIT_OFFSET | VISIT_STRIDE, ndim,
    n1, n2, n3, n4,
    o1, o2, o3, o4,
    s1, s2, s3, s4);
```
3.9 visit_send_string

```c
int visit_send_string(int vcd, int id, double timestamp,
    void *string, visit_type vtype, int size);
```

```fortran
SUBROUTINE FVISIT_SEND_STRING(VCD,ID,TIMESTAMP,STRING,VTYPE,SIZE,
    IERROR)
INTEGER VCD,ID,VTYPE,SIZE4,IERROR
CHARACTER*(*) STRING
REAL*8 TIMESTAMP

$ok =$vcd->send( -id => $id,
    -string => $string,
    [ -timestamp => $timestamp ]
);
```

**Description:**
This function sends a string to the server. The reason to have an extra function for that is that FORTRAN tends to have strange non-portable internal representations of the CHARACTER-datatype. In C, size must be equal to `strlen(data)`, in FORTRAN anything not longer than the declared length of the parameter is allowed. In C you may as well use `visit_send4d` and send a string as a one-dimensional character array.

**Parameters:**
- **vcd**: a valid connection descriptor (or Visit object in Perl).
- **id**: an id that classifies the data. The server side can use this id to identify the data (like the message tag in MPI).
- **timestamp**: a parameter that is intended to further characterize the data, but has no specific meaning.
- **string**: string to be send.
- **vtype**: the datatype of the data, currently only `VISIT_STRING` is allowed, but in later versions other string-like types may be added.
- **size**: number of characters in string (`strlen(data)` in C).

**Return Values:**

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Timed out with no data sent</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>IERROR=1</td>
<td>IERROR=0</td>
<td>IERROR=-1</td>
</tr>
<tr>
<td>Perl</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>
3.10 visit_recv_4d

```c
int visit_recv_4d(int vcd, int id, double *timestamp,
    void *data, visit_type *vtype, int *ndim,
    int *n1, int *n2, int *n3, int *n4, ...);
```

```fortran
SUBROUTINE FVISIT_RECV_4D(VCD, ID, TIMESTAMP, DATA, VTYPE, NDIM,
    N1, N2, N3, N4, IERROR)
```

```fortran
INTEGER VCD, ID, VTYPE, NDIM, N1, N2, N3, N4, IERROR
REAL*8 TIMESTAMP
```

```c
($data, $ok) =
$vcd->recv( -id => $id,
-vtype (&) => $vtype,
    [ -flat => 1, (0) ]
    [ -pack => 1, (0) ]
    [ -dim (&) => [ $n1, $n2, ... ],
      [ -offset (&) => [ $o1, $o2, ... ], (0, 0, ... )
      [ -stride (&) => [ $s1, $s2, ... ], (1, 1, ...) ]
    [ -timestamp (&) => $timestamp ]
    );
```

**Description:**

This is the main client function for receiving data from the server. It sends read–request to the server, asking for certain data. This request contains all the ‘envelope’ information contained in the functions parameters and the id. See 2.5 and 2.4 for a detailed description of datatypes and the ‘envelope’ parameters `vtype, ndim, n1 .. n4, o1 .. o4, s1 .. s4, timestamp`. After receiving this request the server is free to send any data to the client — usually he will fulfill at least parts of the request. To make sure that the received data does not overflow the read buffer (`data`), `visit_recv_4d` guarantees that the size of the received data is not larger than the size of the original request. If the server tries to send more data, `visit_recv_4d` fails and the connection is shut down. When `visit_recv_4d` returns, the envelope parameters are updated with the data send by the server.

In the C–bindings, the `vtype` can be arithmetically or’d with the type modifiers `VISIT_OFFSET` and/or `VISIT_STRIDE`. In that case additional parameters (o1 ... o4 and/or s1 ... s4) have to added to the parameter list. With the FORTRAN bindings you cannot use these optional parameters. Instead, there is an additional function named `visit_recv_4dos` which has fixed offset and stride parameters.
Parameters:

- **vcd**: a valid connection descriptor (or Visit object in Perl).
- **id**: an id that classifies the data. The server can use this id to identify the data (like the message tag in MPI).
- **timestamp**: a parameter that is intended to further characterize the data, but has no specific meaning.
- **data**: pointer to a user supplied receive buffer for the data (in Perl a reference to an array).
- **vtype**: the datatype of the data (see list in 3.7).
- **flat**: indicates that data is a 1D array (Perl only).
- **ndim**: dimensionality of the data, must be 1,2,3, or 4.
- **n1,n2,n3,n4**: size of the data–array, only the first ndim values are used. In Perl, it’s a reference to an array with ndim entries.
- **o1,o2,o3,o4**: optional offsets into the data–array, (see n1..n4 for validity).
- **s1,s2,s3,s4**: optional strides of the data–array, (see n1..n4 for validity).

Return Values:

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Timed out with no data sent</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>IERROR=1 IERROR=0 IERROR=-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Perl</td>
<td>$ok=1 $ok=0, $data=undef $ok=-1, $data=undef</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In C and FORTRAN, the envelope information of the received data is returned in the parameters. In Perl, those parameter names that are marked with (&) have an optional postfix var. If used, the parameter has to be a scalar reference instead of a value. In that case, the information is passed back to the caller, otherwise it is lost (see 2.6).
3.11 visit_recv_4d_os

int visit_recv_4d_os(int vcd, int id, double *timestamp,
    void *data, visit_type *vtype, int *ndim,
    int *n1, int *n2, int *n3, int *n4,
    int *o1, int *o2, int *o3, int *o4,
    int *s1, int *s2, int *s3, int *s4);

SUBROUTINE FVISIT_RECV_4D_OS(VCD,ID,TIMESTAMP,DATA,VTYPE,NDIM,
    N1,N2,N3,N4,O1,O2,O3,O4,S1,S2,S3,S4,
    IERROR)

INTEGER*4 VCD,ID,VTYPE,NDIM,N1,N2,N3,N4,
INTEGER*4 O1,O2,O3,O4,S1,S2,S3,S4,IERROR
REAL*8 TIMESTAMP

Description:
This function has the same functionality as visit_recv_4d. The only difference is that is has a
fixed parameter list including offset and stride.
See visit_recv_4d (3.10) for a description of the parameters and return values. There is no
Perl–binding for this function. The following calls are identical:

visit_recv_4d_os(vcd, id, timestamp, data,
    vtype, ndim,
    n1, n2, n3, n4,
    o1, o2, o3, o4,
    s1, s2, s3, s4);

visit_recv_4d(vcd, id, timestamp, data,
    vtype | VISIT_OFFSET | VISIT_STRIDE, ndim,
    n1, n2, n3, n4,
    o1, o2, o3, o4,
    s1, s2, s3, s4);
3.12 visit_recv_string

int visit_recv_string(int vcd, int id, double *timestamp,
                      void *string, visit_type *vtype, int *size);

SUBROUTINE FVISIT_RECV_STRING(VCD, ID, TIMESTAMP, STRING, VTYPE, SIZE,
                               IERROR)

INTEGER*4    VCD, ID, VTYPE, SIZE, IERROR
REAL*8       TIMESTAMP
CHARACTER*(*) STRING

($string, $ok) = $vcd->recv(   -id   => $id,
                              -size (&) => $size,
                              -vtype => 'STRING',
                              [-timestamp (&) => $timestamp, ]
 );

Description:
This function requests a string from the server. The reason to have an extra function for that is
that FORTRAN tends to have strange non–portable internal representations of the CHARACTER-
datatype. size contains the maximum allowed number of bytes to store in string. In C you
may as well use visit_recv_4d and ask for a one–dimensional character array.

Parameters:
  vcd        a valid connection descriptor (or Visit object in Perl).
  id         an id that classifies the data. The server side can use this id to identify the
data (like the message tag in MPI).
  timestamp  a parameter that is intended to further characterize the data, but has no spe-
cific meaning.
  string     string to be received.
  vtype      the datatype of the data, currently only VISIT_STRING is allowed, but in
              later versions other string–like types may be added.
  size       number of characters in string (strlen(data) in C).

Return Values:

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Timed out with no data sent</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>IERROR=1</td>
<td>IERROR=0</td>
<td>IERROR=-1</td>
</tr>
<tr>
<td>Perl</td>
<td>$ok=1</td>
<td>$ok=0, $data=undef</td>
<td>$ok=-1, $data=undef</td>
</tr>
</tbody>
</table>

In C and FORTRAN, the envelope information of the received data is returned in the parameters.
In Perl, those parameter names that are marked with (&) have an optional postfix var. If used,
the parameter has to be a scalar reference instead of a value. In that case, the information is passed
back to the caller, otherwise it is lost (see 2.6).
Chapter 4

The Visit server API

Implementing a visit–server is typically more complex than including visit–client functions into an application. One reason is that servers don't follow a predefined execution path but have to respond to client requests. This is similar to the way a GUI responds to user actions. In many GUI building toolkits, you register callback functions that shall be executed when certain events occur. The management of events and callbacks is typically performed by a 'mainloop' function which is part of the toolkit and takes control over the application.

A simple visit–server would work in a similar way. In an outer loop, the server would wait for client connections. In an inner loop, the client–requests (identified by their request–id) would have to be dispatched. However, you have to code that loop yourself, since there is no predefined mainloop function in visit. The reason behind this is that a visit–server is intended to be part of a visualization or steering application — and has to cooperate with a GUI toolkit that already has control over the application.

Therefore the GUI must be able to recognize visit–events. Unfortunately, how (and if at all) that can be done depends on both the GUI toolkit and the visit–protocol. Fortunately, there is currently only one visit–protocol, based on TCP/IP sockets. And probably almost every GUI toolkit has the ability to register callbacks for I/O-requests like socket–connect requests or new data being available at a file, pipe, or socket. Pre–build servers that demonstrate how visit works with AVS/Express and Perl/Tk are part of visit. If you want to use other GUI–toolkits you are on your own. However, with the AVS/Express and Perl/Tk code as examples, you should be able to succeed.

Implementing a visit–server that does nothing but dispatching visit–requests is simple. See the test programs vserv.c and vserv.pl for examples. The general procedure is outlined in figure 4.1. The visit–server is started with a call to a protocol–dependent init–function.

Similar to the visit–client API, where only the connect–function is protocol–dependent, the init–function is the only function which is explicitly depending on the protocol. (As noted above, there is an implicit dependence on the protocol, when the server has to respond to requests asynchronously.) After initialization the server is ready to connect to a client. When such a connection has been established (with visit_ssrv_connect) the server can respond to the client’s requests. As discussed in section 2, a request is always initiated by an id sent by the client. This id is followed by a message envelope that contains format information about the message. In case of a send–request, the client sends the data that has been described by the envelope. In case of a receive–request, the server sends back an envelope and message data. When the server is ready to dispatch the next request he informs the client about that by sending an acknowledgment (visit_srv_ack2).

In the rest of this section, all functions of the visit server toolkit are described. We don’t provide FORTRAN language bindings for the server toolkit (who would want to do that ?).
void main()

visit_srv_init_*

until user wants to exit

visit_srv_connect

until disconnected by client

visit_srv_get_id

visit_srv_get_request

send request?

TRUE

FALSE

visit_srv_read_data

visit_srv_write_data

visit_srv_ack2

visit_srv_disconnect

visit_srv_shutdown

Figure 4.1: schematic diagram of a simple visit–server.

4.1 Usage

To use the visit–server toolkit, put one of the following lines of code in your program:

#include "visit_srv.h"    /* C */

use VisitSrv;    # Perl

4.2 visit_srv_init_socket

int visit_srv_init_socket(
    char *service, char *passwd, char *host, int port,
    int flags, int conn_timeout,
    void (*disconnect_cb)(visit_srv_connection *, void *),
    void *disconnect_cb_arg,
    void (*shutdown_cb)(visit_srv_connection *, void *),
    void *shutdown_cb_arg);

$vsrv =
VisitSrv->new(  -service => $servicename,
    -passwd => $passwd,
    [ -host => $host, ('**') ]
    [ -port => $port, (0) ]
    [ -seap_mode => $seapmode, (TOGGLE) ]
    [ -conn_timeout => $conntimeout, (-1)]
    [ -disconnect_cb=> [ \&disconnect_cb, @cb_args ],
    (undef)]
    [ -shutdown_cb => [ \&shutdown_cb, @cb_args ],
    (undef) ]
);
4.2. VISIT_SRV_INIT_SOCKET

Description:
This function creates a socket-based visit–server by opening a listening socket on port port on the interface specified by host. host can be given in any form that is accepted by the gethostbyname function (an IP-address or a name that can be resolved to such an address). With host='*', the systems hostname is used instead. If the port is not available, the port number is increased until a free port is found. When the socket is listening, the host and port are published at the seap–server using the service and passwd parameters. A client only needs to know servicename and passwd to contact the server (see section3.2). The conn_timeout parameter works similarly to the same parameter on the client side. When a request that has been initiated by the client is not finished after conn_timeout seconds, the server assumes that the client is dead and shuts down the connection. There is currently only one feature controlled by the flags parameter. If flags is set to VISIT_SRV_SEAP_TOGGLE, the service is unpublished after a client has connected to the server and published again, after the client has disconnected. Otherwise the service remains published. In any case, the service is unpublished when the server is shut down (by visit_srv_shutdown). In the Perl API, the parameter -seap_mode replaces the flags. Here 'TOGGLE' is default. Use -seap_mode => 0 to switch it off.

The callback parameters are important for the interaction with other tools. If set to non-NULL, they are called when a disconnect or shutdown is performed before anything else is done. This is not only done, when a disconnect/shutdown is called explicitly by the user, but also when this occurs automatically due to an error. This mechanism can be used to unregister socket–events (that may be registered for a GUI toolkit) before the sockets are closed by the visit–server. A visit–server uses two sockets for that purpose. One is listening for connections. It is opened at init–time and closed at shutdown–down. It is listening only when there is no active connection. shutdown_cb will typically be used to remove a callback for that socket. A second socket receives the message ids from the client. disconnect_cb will typically be used to remove a callback for that socket.

Upon success visit_srv_init_socket returns a valid visit–server–descriptor, an integer that is a parameter to all other visit–server functions. For Perl, a VisitSrv object is created.

Parameters:
- service: a service–name that is published by the server.
- passwd: a string that is associated with the service. Unlike the service–name it cannot be queried from the seap–server.
- host: hostname that is published at the seap–server. If set to +"*"+, the callers hostname is used.
- port: initial portnumber for the listening socket. If that port is in use, the number is increased until a free one is found. If port=0, the system chooses a free port.
- flags: if set to VISIT_SRV_SEAP_TOGGLE, the server keeps the service published only while it is not connected to a client.
- conn_timeout: when a request is not finished after conn_timeout seconds, the server assumes that the client is hanging and shuts down the connection.
- disconnect_cb: a function that is called whenever a connection is terminated or breaks.
- disconnect_cb_arg: argument for disconnect_cb.
- shutdown_cb: a function that is called when the server is shut down
- shutdown_cb_arg: argument for shutdown_cb.

Return Values:

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>a non–negative server descriptor</td>
<td>-1</td>
</tr>
<tr>
<td>Perl</td>
<td>a VisitSrv object</td>
<td>undef</td>
</tr>
</tbody>
</table>
4.3 visit_srv_init_socket_raw

int visit_srv_init_socket_raw(char *passwd, char *host, int port,
   int flags, int conn_timeout,
   void (*disconnect_cb)(visit_srv_connection *, void *),
   void *disconnect_cb_arg,
   void (*shutdown_cb)(visit_srv_connection *, void *),
   void *shutdown_cb_arg);

Description:
This function is similar to visit_srv_init_socket (see 4.2). The only difference is that seap is not used. The advantage is that no extra software (the seap–server) is required. However, you have to find another way to communicate the contact information (hostname and port–number) to the clients. Although seap is not used, a passwd has to be specified, because the clients use that passwd to authorize at the server. The parameters host and flags are currently not used. We don’t provide a Perl binding for that function. If you absolutely need one, we leave it as an exercise for you.

Parameters:
passwd a string that is associated with the service. Unlike the service–name it cannot be queried from the seap–server.
host not used.
port initial portnumber for the listening socket. If that port is in use, the number is increased until a free one is found. If port=0, the system chooses a free port.
flags not used.
conn_timeout when a request is not finished after conn_timeout seconds, the server assumes that the client is hanging and shuts down the connection.
disconnect_cb a function that is called whenever a connection is terminated or breaks.
disconnect_cb_arg argument for disconnect_cb.
shutdown_cb a function that is called when the server is shut down
shutdown_cb_arg argument for shutdown_cb.

Return Values:
<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>a non–negative server descriptor</td>
<td>-1</td>
</tr>
<tr>
<td>Perl</td>
<td>a VisitSrv object</td>
<td>undef</td>
</tr>
</tbody>
</table>
4.4 visit_srv_init_file

int visit_srv_init_file(char *filename);

Description:
This creates a pseudo–server. Instead of connecting to clients, this server reads all requests from a file that has been been created with a pseudo–client started by visit_connect_to_file (see section 3.4. Off course, this server type only supports send–requests (it reads client–data from the file). In contrast to a normal server, it does not connect to clients. The server is ready for dispatching requests immediately after the init–call.

Parameters:
filename name of a file that contains pre–recorded visit–messages.

Return Values:
<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>a non–negative server descriptor</td>
<td>-1</td>
</tr>
<tr>
<td>Perl</td>
<td>a VisitSrv object</td>
<td>undef</td>
</tr>
</tbody>
</table>

4.5 visit_srv_connect

int visit_srv_connect(int vsd);

$ok = $vsd->connect();

Description:
This function establishes a connection to a client. It blocks until the connection is available or an error occurs. To be accepted by the server, the client must authorize itself with the correct passwd. The function returns ’1’, when the connections is available, ’0’ if any error has occured. If VISIT_SRV_SEAP_TOGGLE is set, the service will be unpublished at the seap–server after a successful connect.

Parameters:
vsd a valid visit–server descriptor, as returned by visit_srv_init_* (a VisitSrv–object in Perl).

Return Values:
<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, Perl</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
4.6 visit_srv_disconnect

int visit_srv_disconnect(int vsd);

$ok = $vsd->disconnect();

This function closes a connection to a client. Depending on the parameters of the server
visit_srv_init_* call, additional actions will be performed. If set, the disconnect_cb–
function will be called, before anything else is done. If VISIT_SRV_SEAP_TOGGLE is set, the
service will be published again at the seap–server. Upon success, visit_srv_disconnect
returns 1 (it never fails to close a connection).

Parameters:
vsd a valid visit–server descriptor, as returned by visit_srv_init_*
(a VisitSrv–object in Perl).

Return Values:

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, Perl</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

4.7 visit_srv_shutdown

int visit_srv_shutdown(int vsd);

Description:
This function shuts down a visit–server. After that, the server descriptor vsd is no longer valid. It
may be reused by a later call to visit_srv_init_* If set, shutdown_cb will be called, before
anything else is done. If the server has announced his service at a seap–server, he will unpublish it.

Parameters:
vsd a valid visit–server descriptor, as returned by visit_srv_init_*
(a VisitSrv–object in Perl).

Return Values:

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, Perl</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
4.8 visit_srv_get_id

int visit_srv_get_id(int vsd, int *id);

$id = $vsd->get_id();

Description:
This function reads a request–id from a client. It blocks until the id is available or an error occurs.
In case of an error the client is disconnected.

Parameters:
- vsd a valid visit–server descriptor, as returned by visit_srv_init_
  (a VisitSrv–object in Perl).
- id a pointer to the variable that will hold the request–id upon return.

Return Values:

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1, a request–id in id</td>
<td>0</td>
</tr>
<tr>
<td>Perl</td>
<td>a request–id</td>
<td>undef</td>
</tr>
</tbody>
</table>
4.9 visit_srv_get_request

```c
int visit_srv_get_request(int vsd, visit_request *req);
```

```perl
$req = $vsd->get_request();
```

**Description:**
This function returns a new request. This request contains the envelope information (see section 2.4) that is read from the visit–client. In the C–bindings, the request information is stored in a structure named visit_request. This structure contains the fields id, timestamp, vtype, ndim, n1, ... n4, o1, ... o4, s1, ... s4. In Perl, a Visit::Request object is returned that contains the same fields.

See section 2.6 for a description of the Visit::Request object.

visit_srv_get_request blocks until a request is read or an error occurs. In case of an error, the client is disconnected.

**Parameters:**
- **vsd** a valid visit–server descriptor, as returned by visit_srv_init*
- (a VisitSrv–object in Perl).
- **req** a pointer to a request structure that will hold the message envelope upon return.

**Return Values:**

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>size (in bytes) of the message data.</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td>a request in req</td>
<td></td>
</tr>
<tr>
<td>Perl</td>
<td>a Visit::Request object</td>
<td>undef</td>
</tr>
</tbody>
</table>

4.10 visit_srv_read_data

```c
int visit_srv_read_data(int vsd, void *data, visit_request *req);
```

```perl
$data = $req->read( [ -flat => 1 (0) ],
                   [ -pack => 1 (0) ]
                   );
```

**Description:**
This function reads the message data according to the envelope information that is contained in the request stat–structure (or object) req. The function blocks until all data is read or an error occurs. In case of an error, the client is disconnected. In C, data must be a pointer to a buffer that is large enough to hold the requested data. In Perl, a reference to a new array holding the message data is returned. The dimension of the array is as specified by $req %$ or 1, if flat is set to a true value.

**Parameters:**
- **vsd** a valid visit–server descriptor, as returned by visit_srv_init*
- **req** a request structure containing the result of the previous call to visit_srv_get_request,(a Visit::request–object in Perl).
- **flat** data should be returned in a 1D–array (Perl only).
- **data** a pointer to a buffer of sufficient size to hold the message data

**Return Values:**

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1, message in data</td>
<td>0</td>
</tr>
<tr>
<td>Perl</td>
<td>message in $data</td>
<td>undef</td>
</tr>
</tbody>
</table>
4.11 visit_srv_write_data

```c
int visit_srv_write_data(int vsd, void *data,
                        visit_request *req);
```

```perl
$ok = $req->write(
    -data => $data,
    [ -flat => 1 (0) ],
    [ -pack => 1 (0) ]
);
```

**Description:**
This function sends a complete message (envelope and data) to the client. The envelope information is taken from `req`, the data is contained in `data`. The function blocks until all data is sent or an error occurs. In case of an error, the client is disconnected. In C, `data` must be a pointer to a buffer containing the data. In Perl, it is a reference to an array. The dimension of the array must be as specified by `$req %$` or 1, if `flat` is set to a true value.

**Parameters:**
- `vsd` a valid `visit`–server descriptor, as returned by `visit_srv_init_*`
- `req` a request structure containing the result of the previous call to `visit_srv_get_request` (a `Visit::request`–object in Perl).
- `flat` data indicates that is a 1D–array (Perl only).
- `data` a pointer to data (data array in Perl).

**Return Values:**

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, Perl</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
4.12 visit_srv_ack2

int visit_srv_ack2(int vsd);

$ok = $vcd->ack2();

**Description:**
This function sends an acknowledgement to the client which indicates that the server is ready to receive a new client–request. When the client does not receive this acknowledgement within `msg_timeout` seconds after initiating a request, he assumes that the server is not ready to accept this request. In that case, he cancels the request but keeps the connection open. Once the acknowledgement is received by the client, a failure to complete the request within `conn_timeout` will lead the client to close the connection to the server.

A robust `visit`–server should call `visit_srv_ack2` only if he is ready to accept the next request — and not when he has just read the data from the last request.

Upon success, `visit_srv_ack2` returns 1. If an error occurs, it returns 0 and disconnects the client.

**Parameters:**
- **vsd** a valid `visit`–server descriptor, as returned by `visit_srv_init_*` (a `VisitSrv`–object in Perl).

<table>
<thead>
<tr>
<th>Return Values:</th>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, Perl</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

4.13 visit_srv_configure

int visit_srv_configure(int vsd, int what, ...);

$vsd->configure(-parameter_name => $new_value);

int visit_srv_configure(int vsd, int what, ...);

**Description:**
This function can be used to modify properties of an active `visit`–server. Some of the parameters are specific to a protocol, some are generic. The parameter `what` is an integer that specifies which property shall be modified. Symbolic names for these integers are defined in the `visit_srv.h` header file. The next parameter is the new value of that property. Currently, the following properties can be modified.
4.14 VISIT_SRV_SOCKET_LSD, VISIT_SRV_SOCKET_CSD,

On socket–connections:

<table>
<thead>
<tr>
<th>what</th>
<th>property/parameter to change</th>
<th>type of parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>VISIT_SERVICE</td>
<td>service</td>
<td>string</td>
</tr>
<tr>
<td>VISIT_PASSWD</td>
<td>passwd</td>
<td>string</td>
</tr>
<tr>
<td>VISIT_HOST</td>
<td>host</td>
<td>string</td>
</tr>
<tr>
<td>VISIT_PORT</td>
<td>port</td>
<td>integer</td>
</tr>
<tr>
<td>VISIT_CONN_TIMEOUT</td>
<td>conn_timeout</td>
<td>integer</td>
</tr>
</tbody>
</table>

If a seap–related parameter is changed, while a server has its service published, these changes are immediately transmitted to the seap–server.

No parameters can be configured for file–connections.

The Perl binding uses the names of the properties/parameters (as in the new method) to change instead of the integer what.

Parameters:

- vsd: a valid visit–server descriptor, as returned by visit_srv_init_*(a VisitSrv–object in Perl).
- what: an integer value, specifying the parameter to change.
- ... the new value of that parameter.

Return Values:

<table>
<thead>
<tr>
<th>Binding</th>
<th>Success</th>
<th>Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>C, Perl</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

4.14 visit_srv_socket_lsd, visit_srv_socket_csd,

int visit_srv_socket_lsd(int vsd);
int visit_srv_socket_csd(int vsd);

Description:

These functions are specific for the socket-based visit–server. They return the socket–descriptors of the listening socket (_lsd) or the socket that receives the message ids (_csd). Events on these sockets (accept or data available respectively) will trigger actions in typical visit–servers. To achieve that, the socket descriptors usually need to be registered at the GUI-toolkit (or explicitely used in a select–call). See 4.2 for details.

Parameters:

- vsd: a valid visit–server descriptor, as returned by visit_srv_init_*(a VisitSrv–object in Perl).

Return Values:

The functions return a valid socket descriptor or -1 if the socket is not active.
Chapter 5

The AVS/Express visit–server

This API implements three macros for visit. The module visitserver controls the connection to a visit client program. The module visitreader and visitwriter are responsible for the data transfer from/to the visit client program.

Figure 5.1: AVS/Express network which uses the visit macros and the Panel of the visitserver macro.

The visitserver panel contains text widgets for the information which will be published at the seap server, a status line which describes the actual state of the visit connection and a button which switches the connection active or inactive. Whenever a request arrives the visitserver delegates the request to the corresponding visitreader/visitwriter. Therefore the trigger output port of visitserver contains the id of actual request. The multiplexer macro which is also included in the visit package activates the corresponding output port and a visitreader/visitwriter which is connected to this port. This macro reads or writes the data from/to the connection.

The four macros visitserver, visitreader, visitwriter and multiplexer are only wrappers for the corresponding modules VisitServer, VisitReader, VisitWriter and Multiplexer which are implemented in the programming language C. The following sections describes this macros and modules. The section 8 (Demos) contains two examples for the AVS/Express visit–server.

These visit macros and modules are published at the International AVS Centre\(^1\) (IAC) under the project name Visit. The modules can be found in the Folder Data IO. The two examples described in the section 8 (Demos) are located in the folder Examples of the IAC library.

\(^1\)URL: http://www.iavsc.org/
5.1 visitserver

This functional macro controls the connection to remote applications which uses the VISIT library.

Parameters

The following lists all of the parameters found in the parameter block VisitServerParams, which are accessed by the module VisitServer and the UI Macro VisitUI.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>UI Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>SeapService</td>
<td>string</td>
<td>Service name under which the visualization will be announced at the SEAP-Server</td>
<td>UItext</td>
</tr>
<tr>
<td>SeapPasswd</td>
<td>string</td>
<td>password for checking the service at SEAP-server</td>
<td>UItext</td>
</tr>
<tr>
<td>Interface</td>
<td>string</td>
<td>host-name that is published at SEAP-server. If set to &quot;*&quot; the callers hostname is used</td>
<td>UItext</td>
</tr>
<tr>
<td>Listen</td>
<td>int</td>
<td>enables/disables listening socket for connecting to an applications</td>
<td>UItoggle</td>
</tr>
<tr>
<td>IdDescriptions</td>
<td>string[]</td>
<td>data description string for corresponding id</td>
<td>-</td>
</tr>
</tbody>
</table>

Output Ports

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SockID</td>
<td>int</td>
<td>socket descriptor of the data-connection to the remote applications, needed by the VisitReader module</td>
</tr>
<tr>
<td>Trigger</td>
<td>int</td>
<td>value of the ID-parameter of the actual client request; This port should be connected with the input port of the multiplexer module. This module activates one of its own output ports and a VisitReader or -Writer module connected to it.</td>
</tr>
<tr>
<td>Action</td>
<td>int</td>
<td>describes the status of the connection to a remote application: 0: not listening (Listen==0), 1: listening, but no connected, 2: connected</td>
</tr>
<tr>
<td>Status</td>
<td>string</td>
<td>describes the status of the connection</td>
</tr>
</tbody>
</table>

Description

This macro controls the connection to remote applications. If the flag Listen is on, it establishes a socket with the next free port number above the value of the internal parameter Port and announces the information about this Port at the SEAP-server under the service-name and password from SeapService and SeapPasswd.

Then it waits for a connection to the announced port. After connecting to a remote application, the macro waits for requests on this connection. The output port SockId is set to the socket descriptor of this connection. Each request contains an Id which determines which VisitReader or -Writer should process the request. The output port Trigger will be set to this value. This port should be connect with the input port of the multiplexer macro. Depending on the value of Trigger, the multiplexer activates one of its output ports which in turn activates the visitreader or visitwriter macro connected to it. This module then reads the request data from the socket.

After a client has connected, the service is deleted from the SEAP-Server. Each visitserver can control only one connection at a time. Therefore, each time a new connection request comes in, the old connection is shut down. If a connection is shut down by the remote application, visitserver announces its service again.
The visitserver macro should be used in conjunction with the macros multiplexer, visitreader and visitwriter.

**Inputs**

**SeapService, SeapPasswd, Interface**: Service name and password under which the visualization will be announced at the SEAP-Server. The SEAP-Server runs on a different machine and stores entries which describe services of visualization applications. Remote applications can ask the SEAP-Server for such services and then receive the Portnumber of the socket and the hostname on which the listening socket is established from the SEAP-Server. This additional process removes the need to use hard-coded Interface names and port numbers. It is also possible to change the visualization workstation while the remote application keeps on running.

**Listen**: This flag enables/disables the listening socket. Only if the flag is set, a listening socket will be established. A running connection will be stopped if the flag is set to off.

**IdDescriptions**: The messages contain a message id, which determines which VisitReader or -Writer should process the message. In the string array IdDescriptions a description string can be assigned to the message Id. This string will be used in status messages in the VisitUI or in stdout messages.

**Outputs**

**SockID**: SockID is the socket descriptor of the data-connection to the remote application. The macros visitreader and visitwriter need this SockId to read the data from the socket.

**Trigger**: Trigger is the value of the ID-parameter of the message. It is intended to be used to transport data to different AVS-modules by the help of the multiplexer macro.

**Action**: This port describes the status of the connection to a remote application: 0: not listening (Listen==0), 1: listening, but no connected, 2: connected. This port can be used to switch the color of a status display like a traffic light: 0: red, 1: yellow, 2: green

**Status**: This output port describes the status of the connection. The String contains the action (not listening, listening, connected), the id of last message received and the overall number of messages received while connected.

**Utility Modules**

The User Macro visitserver combines the functional Macro VisitServer with the UI Macro VisitServerUI. The User macro contains the parameter block VisitServerParams and an initialized string array iddescriptions.

**Example**

An example application VisitGoLEg is provided that works together with the remote application cgol (game of life) of the VISIT-library distribution (demo/visit_sim). The cgol application computes the game of life for a 3d field, which will be sent to AVS/Express every life step. It is possible to insert new blocks at selectable positions, and to stop and suspend the remote application.

**Files**

`iac_proj/visit/visit_macs.v` contains the V definitions of the functional macro visitserver, the UI macro VisitServerUI and the example application VisitEg.
Prerequisites

For using visitserver and to run the demo application the VISIT-library must be installed and also a SEAP-server must be running. For installation of these tools see this manual or http://www.fz-juelich.de/zam/visit.

Other Notes

The VisitMacs library inherits its process. As this library contains no procedural code, the process is not important. The modules in the low-level VisitMods library execute under the process specified in that library, not the process defined in the high-level library. By default the express process will be used.

5.2 visitreader

This functional macro reads data from a visit connection.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SockID</td>
<td>int</td>
<td>Socket descriptor of the data-connection to the remote application.</td>
</tr>
<tr>
<td>Trigger</td>
<td>int</td>
<td>visitreader reads data from the data socket if this port is activated.</td>
</tr>
</tbody>
</table>

Output Ports

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeStamp</td>
<td>double</td>
<td>Timestamp of the data send to the visualization</td>
</tr>
<tr>
<td>n1</td>
<td>int</td>
<td>size of field in dimension 1</td>
</tr>
<tr>
<td>n2</td>
<td>int</td>
<td>size of field in dimension 2</td>
</tr>
<tr>
<td>n3</td>
<td>int</td>
<td>size of field in dimension 3</td>
</tr>
<tr>
<td>n4</td>
<td>int</td>
<td>size of field in dimension 4</td>
</tr>
<tr>
<td>DataInt</td>
<td>int[]</td>
<td>data field, if datatype of message is INT32</td>
</tr>
<tr>
<td>DataShort</td>
<td>short[]</td>
<td>data field, if datatype of message is INT16</td>
</tr>
<tr>
<td>DataByte</td>
<td>byte[]</td>
<td>data field, if datatype of message is BYTE</td>
</tr>
<tr>
<td>DataDouble</td>
<td>double[]</td>
<td>data field, if datatype of message is FLOAT64</td>
</tr>
<tr>
<td>DataString</td>
<td>string</td>
<td>data field, if datatype of message is STRING</td>
</tr>
<tr>
<td>DataIntScalar</td>
<td>int</td>
<td>data, if datatype is INT32 and the field length is 1</td>
</tr>
<tr>
<td>DataShortScalar</td>
<td>short</td>
<td>data, if datatype is INT16 and the field length is 1</td>
</tr>
<tr>
<td>DataByteScalar</td>
<td>byte</td>
<td>data, if datatype is BYTE and the field length is 1</td>
</tr>
<tr>
<td>DataDoubleScalar</td>
<td>double</td>
<td>data, if datatype is FLOAT64 and the field length is 1</td>
</tr>
</tbody>
</table>

Description

This macro reads data from a remote application. It needs a SockID that is provided by a visitserver macro for a connection. The macro is triggered whenever data arrives at the Trigger port. Trigger can be connected directly to the Trigger output port of a visitserver. If more than one visitreader or visitwriter is connected to a visitserver it is necessary to use a multiplexer in order to select one
visitreader or visitwriter macro for each request. There is no implicit distinction between read and write requests. Therefore the user is responsible for using different IDs for read and write requests. The values TimeStamp and n1 ... n4 contain the header-information that has been send by the remote application (using the visit_send_4d or visit_send_string function call). Depending on the type of data that has been send the data is presented on the appropriate Data output port. With a field of length n1=n2=n3=n4=1 the data is presented both at a vector and a scalar output port.

The visitreader macro should be used in conjunction with the macros multiplexer, visitserver and visitwriter.

Inputs

SockID: Socket descriptor of the data-connection to the remote applications. This input port should be connected with the output port SockID of visitserver.

Trigger: This port should be connected with an output port of the multiplexer macro which is connected with the Trigger port of the visitserver macro. This port will be activated if a message is arrived for this reader.

Outputs

Timestamp, n1, n2, n3, n4: Contain the header-information that has been send by the remote application (using the visit_send_4d or visit_send_string function call).

DataInt, DataShort, DataByte, DataDouble, DataString, DataIntScalar, DataShortScalar, DataByteScalar, DataDoubleScalar:
Depending on the datatype of the message one (or two) of these output ports present the data of the message (see table of output ports above).

Utility Modules

The User Macro visitreader is only a wrapper macro for the module VisitReader. All input and output ports of this Module are connected with the macro parameters.

Example

An example application VisitGoLEg is provided that works together with the remote application cgol (game of life) of the VISIT-library distribution (demo/visit_sim). The cgol application computes the game of life for a 3d field, which will be sent to AVS/Express every life step. It is possible to insert new blocks at selectable positions, and to stop and suspend the remote application.

Files

iac_proj/visit/visit_macs.v contains the V definitions of the functional macro visitreader.

Prerequisites

For using visitreader and to run the demo application the VISIT-library must be installed and also a SEAP-server must be running. For installation of these tools see http://www.fz-juelich.de/zam/visit.

Other Notes

The VisitMacs library inherits its process. As this library contains no procedural code, the process is not important. The modules in the low-level VisitMods library execute under the process specified in that library, not the process defined in the high-level library. By default the express process will be used.
5.3 visitwriter

This functional macro writes data to a visit connection.

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>UI</th>
</tr>
</thead>
<tbody>
<tr>
<td>SockID</td>
<td>int</td>
<td>Socket descriptor of the data-connection to the remote application.</td>
<td></td>
</tr>
<tr>
<td>Trigger</td>
<td>int</td>
<td>visitwriter writes data to the data socket if this port is activated.</td>
<td></td>
</tr>
<tr>
<td>TimeStamp</td>
<td>double</td>
<td>Timestamp of the data send to the visualization</td>
<td></td>
</tr>
<tr>
<td>n1</td>
<td>int</td>
<td>size of field in dimension 1</td>
<td></td>
</tr>
<tr>
<td>n2</td>
<td>int</td>
<td>size of field in dimension 2</td>
<td></td>
</tr>
<tr>
<td>n3</td>
<td>int</td>
<td>size of field in dimension 3</td>
<td></td>
</tr>
<tr>
<td>n4</td>
<td>int</td>
<td>size of field in dimension 4</td>
<td></td>
</tr>
<tr>
<td>DataInt</td>
<td>int[]</td>
<td>data field, if datatype of message is INT32</td>
<td></td>
</tr>
<tr>
<td>DataShort</td>
<td>short[]</td>
<td>data field, if datatype of message is INT16</td>
<td></td>
</tr>
<tr>
<td>DataByte</td>
<td>byte[]</td>
<td>data field, if datatype of message is BYTE</td>
<td></td>
</tr>
<tr>
<td>DataDouble</td>
<td>double[]</td>
<td>data field, if datatype of message is FLOAT64</td>
<td></td>
</tr>
<tr>
<td>DataString</td>
<td>string</td>
<td>data field, if datatype of message is STRING</td>
<td></td>
</tr>
<tr>
<td>DataIntScalar</td>
<td>int</td>
<td>data, if datatype is INT32 and the field length is 1</td>
<td></td>
</tr>
<tr>
<td>DataShortScalar</td>
<td>short</td>
<td>data, if datatype is INT16 and the field length is 1</td>
<td></td>
</tr>
<tr>
<td>DataByteScalar</td>
<td>short</td>
<td>data, if datatype is BYTE and the field length is 1</td>
<td></td>
</tr>
<tr>
<td>DataDoubleScalar</td>
<td>double</td>
<td>data, if datatype is FLOAT64 and the field length is 1</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

On request, this macro sends data back to a remote application. Like the visitreader this macro is triggered by the Trigger port. The Trigger is activated by a request from the remote application (visit_recv_4d or visit_recv_string) when SockID and Trigger are connected to a visitserver (via multiplexer). The remote application asks for a specific datatype with specific array dimensionality and bounds. The datatype of the request is used to select the data to be send from the various input ports. The array dimensionality and bounds are only used to make sure that the macro does not send more data than the remote application expects. If the values at n1 ... n4 are connected or set to something not equal to -1, those values are used for the transfer. Otherwise the values in the original request remain unchanged if n1*n2*n3*n4 matches the size of the array at the input port or are set to n1=n2=n3=n4=1. n1 is set to the size of the array at the input port in the latter case. If ndim=n1=1, the data is taken from a scalar input port.

This sounds obscure to you? It is! Just make sure, that the AVS application always provides the data that the remote application expects and you don’t have to worry about the parameters n1 to n4 at all.

The visitwriter macro should be used in conjunction with the macros multiplexer, visitserver and visitreader.

**Inputs**

**SockID:** Socket descriptor of the data-connection to the remote applications. This input port should be connected with the output port SockID of visitserver.

**Trigger:** This port should be connected with an output port of the multiplexer macro which is connected with the Trigger port of the visitserver macro. This port will be activated if a request is arrived for this writer.
**Timestamp, n1, n2, n3, n4:** Contain the header-information that will be send to the remote application.

**DataInt, DataShort, DataByte, DataDouble, DataString, DataIntScalar, DataShortScalar, DataByteScalar, DataDoubleScalar:**
Depending on the datatype of the message one of these input ports gives the data of the message.

**Utility Modules**

The low-level VisitWriter module is used in the functional macro visitwriter.

**Example**

An example application VisitGoLEg is provided that works together with the remote application cgol (game of life) of the VISIT-library distribution (demo/visit_sim). The cgol application computes the game of life for a 3d field, which will be sent to AVS/Express every life step. It is possible to insert new blocks at selectable positions, and to stop and suspend the remote application.

**Files**

iac_proj/visit/visit_macs.v contains the V definitions of the functional macro visitwriter.

**Prerequisites**

For using visitwriter and to run the demo application the VISIT-library must be installed and also a SEAP-server must be running. For installation of these tools see http://www.fz-juelich.de/zam/visit.

**Other Notes**

The VisitMacs library inherits its process. As this library contains no procedural code, the process is not important. The modules in the low-level VisitMods library execute under the process specified in that library, not the process defined in the high-level library. By default the express process will be used.
5.4 multiplexer

A multiplexer for integer numbers.

Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>UI Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inval</td>
<td>int</td>
<td>number which indicates which output port should be activated.</td>
<td></td>
</tr>
</tbody>
</table>

Output Ports

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out1 ... Out10</td>
<td>int</td>
<td>Out&lt;\textit{num}&gt; will be activated if Inval==$num$</td>
</tr>
</tbody>
</table>

Description

This simple module is used to trigger one of its output ports whenever it receives something on its input port. If an integer <$\textit{num}$> between 1 and 10 arrives at ’Inval’, it is passed to the output Port Out<$\textit{num}>$.

Hint: if your application requires more than 10 IDs you can use multiple ’Multiplexer’s and specify the connection to the ’VisitServer’ with ’Trigger-10’ (or similar). The ’Multiplexer’ then acts on trigger values 11 to 20.

Inputs and Outputs

\textit{Inval}: This port gives the number which indicates which output port should be activated.

\textit{Out1 ... Out10}: Depending on the value of the input parameter ’Inval’ one of these outputs port will be activated.

Utility Modules

The User Macro multiplexer is only a framework for the module Multiplexer. All input and output ports of this Module are connected with the macro parameters.

Example and Files

An example for this macro can be found in VisitGoLEg. \textit{iac\_proj/visit/visit\_macs.v} contains the V definitions of the functional macro multiplexer.

Other Notes

The VisitMacs library inherits its process. As this library contains no procedural code, the process is not important. The modules in the low-level VisitMods library execute under the process specified in that library, not the process defined in the high-level library. By default the express process will be used.
5.5 VisitServer

This module controls the connection to remote applications which uses the VISIT library.

```c
module VisitServer<src_file="VisitServer.c",process="express"> {
  omethod+notify_inst+req VisitServer_inst (Port+read+req,Interface+read+req,SeapService+read+req,
    SeapPasswd+read+req, Listen+read+req,Action+write, Status+write ) = "VisitServer_inst";
  omethod+notify_deinst VisitServer_deinst( Action+write, Status+write ) = "VisitServer_deinst";
  omethod+req SeapUpdateService( SeapService+read+notify+req, Action+write,Status+write ) = "SeapUpdateService";
  omethod+req SeapUpdatePasswd( SeapPasswd+read+notify+req, Action+write, Status+write ) = "SeapUpdatePasswd";
  omethod+req SeapUpdateInterface( Interface+read+notify+req, Action+write, Status+write ) = "SeapUpdateInterface";
  omethod+req ListenUpdate( Listen+read+notify+req, Action+write, Status+write ) = "ListenUpdate";
  omethod+req IdDescriptionsUpdate( IdDescriptions+read+notify ) = "IdDescriptionsUpdate";
  int Port<NEportLevels={0,0}> = 0;
  string Interface<NEportLevels={2,0}> = "*";
  int Listen<NEportLevels={2,0}> = 0;
  string SeapService<NEportLevels={2,0}> = "VISIT_AVS";
  string SeapPasswd<NEportLevels={2,0}> = "demo";
  int SockID<NEportLevels={0,2}> = -1;
  int Trigger<NEportLevels={0,2}> = 0;
  int Action<NEportLevels={0,2}> = 0;
  string Status<NEportLevels={0,2}> = "<Init>";
  string IdDescriptions<NEportLevels={2,0}>=[];
  ptr internal<NEportLevels={0,0}>;
};
```

Description

This module controls the connection to remote applications. If the flag Listen is on, it establishes a socket with the next free port number above the value of the internal parameter Port and announces the information about this Port at the SEAP-server under the service-name and password from SeapService and SeapPasswd. Then it waits for a connection to the announced port. After connecting to a remote application, the module waits for requests on this connection. The output port SockId is set to the socket descriptor of this connection. Each request contains an Id which determines which VisitReader or -Writer should process the request. The output port Trigger will be set to this value. This port should be connect with the input port of the multiplexer module. Depending on the value of Trigger, the multiplexer activates one of its output ports which in turn activates the VisitReader or -Writer module connected to it. This module then reads the request data from the socket. After a client has connected, the service is deleted from the SEAP-Server. Each VisitServer can control only one connection at a time. Therefore, each time a new connection request comes in the old connection is shut down. If a connection is shut down by the remote application, VisitServer announces its service again.

The VisitServer module should be used in conjunction with the modules Multiplexer, VisitReader and VisitWriter.

By default the low-level library visit_mods, which needs to be compiled, has the process set to express.
5.5. VISITSERVER

**Inputs**

*SeapService, SeapPasswd, Interface:* Service name and password under which the visualization will be announced at the SEAP-Server. The SEAP-Server runs on a different machine and stores entries which describe services of visualization applications. Remote applications can ask the SEAP-Server for such services and then receive the Portnumber of the socket and the hostname on which the listening socket is established from the SEAP-Server. This additional process removes the need to use hard-coded Interface names and port numbers. It is also possible to change the visualization workstation while the remote application keeps running.

**Listen:** This flag enables/disables the listening socket. Only if the flag is set, a listening socket will be established. A running connection will be stopped if the flag is set to off.

**IdDescriptions:** The messages contain a message id, which determines which VisitReader or -Writer should process the message. In the string array IdDescriptions a description string can be assigned to the message Id. This string will be used in status messages in the VisitUI or in stdout messages.

**Outputs**

*SockID:* SockID is the socket descriptor of the data-connection to the remote application. The macros visitreader and visitwriter need this SockId to read the data from the socket.

**Trigger:** Trigger is the value of the ID-parameter of the message. It is intended to be used to transport data to different AVS-modules by the help of the multiplexer macro.

**Action:** This port describes the status of the connection to a remote application: 0: not listening (Listen==0), 1: listening, but no connected, 2: connected. This port can be used to switch the color of a status display like a traffic light: 0: red, 1: yellow, 2: green

**Status:** This output port describes the status of the connection. The String contains the action (not listening, listening, connected), the id of last message received and the overall number of messages received while connected.

**Utility Modules**

The low-level VisitServer module is used in the functional macro visitserver.

**Example and Files**

An example for this macro can be found in VisitGoLeg. *iac_proj/visit/visit mods.v* contains the V definitions of the VisitServer module.
5.6 VisitReader

This functional macro reads data from a visit connection.

```c
module VisitReader<src_file="VisitReader.c",process="express"> {
  omethod+req VisitReader_read(
    SockID+read+req,Trigger+read+notify+req,DataDoubleSize+write,
    DataDouble+write,TimeStamp+write,n1+write,n2+write,n3+write,
    n4+write,DataIntSize+write,DataInt+write,DataShortSize+write,
    DataShort+write,DataByteSize+write,DataByte+write,
    DataString+write,DataIntScalar+write,DataShortScalar+write,
    DataByteScalar+write,DataDoubleScalar+write
  ) = "VisitReader_read";
  int SockID<NEportLevels={2,0}> = -1;
  int Trigger<NEportLevels={2,0}> = 0;
  double TimeStamp<NEportLevels={0,2}>;
  int n1<NEportLevels={0,2}> = -1;
  int n2<NEportLevels={0,2}> = -1;
  int n3<NEportLevels={0,2}> = -1;
  int n4<NEportLevels={0,2}> = -1;
  int o1<NEportLevels={0,0}> = -1;
  int o2<NEportLevels={0,0}> = -1;
  int o3<NEportLevels={0,0}> = -1;
  int o4<NEportLevels={0,0}> = -1;
  int s1<NEportLevels={0,0}> = -1;
  int s2<NEportLevels={0,0}> = -1;
  int s3<NEportLevels={0,0}> = -1;
  int s4<NEportLevels={0,0}> = -1;
  int DataIntSize = 0;
  int DataInt<NEportLevels={0,2}>[DataIntSize];
  int DataShortSize = 0;
  short DataShort<NEportLevels={0,2}>[DataShortSize];
  int DataByteSize = 0;
  byte DataByte<NEportLevels={0,2}>[DataByteSize];
  int DataDoubleSize = 0;
  double DataDouble<NEportLevels={0,2}>[DataDoubleSize];
  string DataString<NEportLevels={0,2}>;
  int DataIntScalar<NEportLevels={0,2}>;
  short DataShortScalar<NEportLevels={0,2}>;
  byte DataByteScalar<NEportLevels={0,2}>;
  double DataDoubleScalar<NEportLevels={0,2}>;
  omethod+notify_inst VisitReader_inst(
    n1+read+write,TimeStamp+write,n2+read+write,n3+read+write,
    n4+read+write,DataIntSize+write,DataInt+write,DataShortSize+write,
    DataShort+write,DataByteSize+write,DataByte+write,
    DataString+write,DataIntScalar+write,DataShortScalar+write,
    DataByteScalar+write,DataDoubleScalar+write
  ) = "VisitReader_inst";
};
```
5.7. VISITWRITER

Description
This module reads data from a remote application. It needs a SockID that is provided by a VisitServer module for a connection. The module is triggered whenever data arrives at the Trigger port. Trigger can be connected directly to the Trigger output port of a VisitServer. If more than one VisitReader or VisitWriter is connected to a VisitServer it is necessary to use a Multiplexer in order to select one VisitReader or VisitWriter module for each request. There is no implicit distinction between read and write requests. Therefore the user is responsible for using different IDs for read and write requests.

The values TimeStamp and n1 ... n4 contain the header-information that has been send by the remote application (using the visit_send4d or visit_send_string function call). Depending on the type of data that has been send the data is presented on the appropriate Data output port. With a field of length n1=n2=n3=n4=1 the data is presented both at a vector and a scalar output port.

The VisitReader macro should be used in conjunction with the modules Multiplexer, VisitServer and VisitWriter.

Inputs

SockID: Socket descriptor of the data-connection to the remote applications. This input port should be connected with the output port SockID of visitserver.

Trigger: This port should be connected with an output port of the multiplexer macro which is connected with the Trigger port of the visitserver macro. This port will be activated if a message is arrived for this reader.

Outputs

Timestamp, n1, n2, n3, n4: Contain the header-information that has been send by the remote application (using the visit_send4d or visit_send_string function call).

DataInt, DataShort, DataByte, DataDouble, DataString, DataIntScalar, DataShortScalar, DataByteScalar, DataDoubleScalar:
Depending on the datatype of the message one (or two) of these output ports present the data of the message (see table of output ports above).

Utility Modules

The low-level module VisitReader is used in the functional macro visitreader.

Example and Files

An example for this module can be found in VisitGoLEg. iac_proj/visit/visit mods.v contains the V definitions of the VisitReader module.

Other Notes

By default the low-level library visit mods, which needs to be compiled, has the process set to express.

5.7 VisitWriter

This module writes data to a visit connection.

module VisitWriter<src_file="VisitWriter.c",process="express"> {
On request, this module sends data back to a remote application. Like the VisitReader this module is triggered by the Trigger port. The Trigger is activated by a request from the remote application (visit_recv_4d or visit_recv_string) when SockID and Trigger are connected to a VisitServer (via multiplexer). The remote application asks for a specific datatype with specific array dimensionality and bounds. The datatype of the request is used to select the data to be send from the various input ports. The array dimensionality and bounds are only used to make sure that the module does not send more data than the remote application expects. If the values at n1 ... n4 are connected or set to something not equal to -1, those values are used for the transfer. Otherwise the values in the original request remain unchanged if n1*n2*n3*n4 matches the size of the array at the input port or are set to n1=n2=n3=n4=1. n1 is set to the size of the array at the input port in the latter case. If ndim=n1=1, the data is taken from a scalar input port.

This sounds obscure to you? It is! Just make sure, that the AVS application always provides the data that the remote application expects and you don’t have to worry about the parameters n1 to n4 at all.

The VisitWriter module should be used in conjunction with the modules Multiplexer, VisitServer and VisitReader.

**Inputs**

**SockID:** Socket descriptor of the data-connection to the remote applications. This input port should be connected with the output port SockID of visitserver.
Trigger: This port should be connected with an output port of the multiplexer macro which is connected with the Trigger port of the visitserver macro. This port will be activated if a request is arrived for this writer.

Timestamp, n1, n2, n3, n4: Contain the header-information that will be send to the remote application.

DataInt, DataShort, DataByte, DataDouble, DataString, DataIntScalar, DataShortScalar, DataByteScalar, DataDoubleScalar: Depending on the datatype of the message one of these input ports gives the data of the message.

Utility Modules

The low-level module VisitWriter is used in the functional macro visitwriter.

Example and Files

An example for this module can be found in VisitGoLEg. `iac_proj/visit/visit mods.v` contains the V definitions of the VisitWriter module.

Other Notes

By default the low-level library visit mods, which needs to be compiled, has the process set to express.
5.8 Multiplexer

A multiplexer for integer numbers.

```vhdl
module Multiplexer<src_file="Multiplexer.c",process="express"> {
  omethod+req Multiplex( Inval+read+notify+req,Out1+write,
  Out2+write,Out3+write,Out4+write,Out5+write,Out6+write,
  Out7+write,Out9+write,Out10+write)="Multiplex";
  int Inval<NEportLevels={2,0}>;
  int Out1<NEportLevels={0,2}>;
  int Out2<NEportLevels={0,2}>;
  int Out3<NEportLevels={0,2}>;
  int Out4<NEportLevels={0,2}>;
  int Out5<NEportLevels={0,2}>;
  int Out6<NEportLevels={0,2}>;
  int Out7<NEportLevels={0,2}>;
  int Out9<NEportLevels={0,2}>;
  int Out10<NEportLevels={0,2}>;
};
```

**Description**

This simple module is used to trigger one of its output ports whenever it receives something on its input port. If an integer \(<num>\) between 1 and 10 arrives at 'Inval', it is passed to the output Port Out\(<num>\).

Hint: if your application requires more than 10 IDs you can use multiple 'Multiplexer's and specify the connection to the 'VisitServer' with 'Trigger-10' (or similar). The 'Multiplexer' then acts on trigger values 11 to 20.

**Inputs and Outputs**

*Inval*: This port gives the number which indicates which output port should be activated.

*Out\(1\) ... Out\(10\)*: Depending on the value of the input parameter 'Inval' one of these outputs port will be activated.

**Utility Modules**

The low-level Multiplexer module is used in the functional macro multiplexer.

**Example and Files**

An example for this module can be found in VisitGoLeg. `iac_proj/visit/visit_mods.v` contains the V definitions of the Multiplexer module.

**Other Notes**

By default the low-level library `visit_mods`, which needs to be compiled, has the process set to `express`. 
Chapter 6

seap — the service announcement protocol

seap is an acronym for service announcement protocol. The idea is that if a client and a server want to get into contact, they either need to agree about a contact point in advance or they need a third party to exchange that information. The 'seap_server' supplies this service using the 'service announcement protocol'. A visit–server application announces a service by telling the 'seap_server' a 'service name', a 'passwd' and where this service can be reached (hostname and portnumber). A client-application can query this information if it knows the service name and the passwd.

Why reinvent the wheel once more? Well, seap has very limited functionality, has no access control, almost no security, and no persistence incorporated, but it also is a very thin layer: the server has only about 220 lines of perl-code, the client only some 140 lines of C-code. Thus, SEAP can easily be ported and incorporated in any non-real-world-production (of course!) code.

seap consists of:

- **seap_server**: a Perl-implementation of the seap–server.
- **seap**: a Perl/Tk-based client that can be used to monitor the state of the seap_server.
- **seap.c**: the C-client functions (part of the visit-library).
- **Seap.pm**: a Perl module with seap client functions.
- **sclient**: a demo-client program.
- **visitrc**: a system–wide configuration file that tells clients where the server is located.
- **$HOME/.visitrc**: an optional private configuration file that can override visitrc

The location of the seap–server is taken from a config file named visitrc that is installed in the same directory as the visit–library libvisit.a. The hostname where the seap–server is running as well as the port–number where the server accepts requests are taken from that file. The format is as follows:

```
seap_server : seapserver.mydomain.de
seap_port : 4711
```

The values in this file can be overridden by a file named .visitrc in the user’s home–directory.
6.1 The seap—server

To use seap, you need to have the seap_server running at least during the runtime of both the visit—client and —server application. The typical mode of operation is to have the seap—server permanently running on a machine at your site, so that clients can register and query services all the time. The shell—script check_seap_server should be run periodically from cron to monitor the seap—server and restart it if needed.

A normal user can query or unpublish only those services for which he knows the passwd. However, the seap—server has a ’master—passwd’ that can be used to query and unpublish any service. But even with this master—passwd it is not possible to obtain other passwds.

The seap_server reads its parameters from a configuration file. The name of the file can be given on the command line (default is $HOME/.visitserverrc.) The format of the config file is as follows:

```plaintext
seap_port : 4711
seap_pidfile : /tmp/seap_server.pid
seap_passwd : master_passwd
seap_debug : 1
```

These values have the following meaning:

- **seap_port**: the port number of the port where clients connect to the server.
- **seap_pidfile**: the name of a file, where the server stores its process id. The shell script check_seap_server uses this file to check whether the server is running or has to be restarted.
- **seap_passwd**: the master—passwd.
- **seap_debug**: this parameter is optional. If set to non-zero, the server prints logging messages (including passwords !) to stdout. So, please use it only for debugging.

6.2 The seap client functions

The seap client API provides functions for publishing, unpublishing, and querying services at a seap—server. We only provide bindings for C and Perl. The Perl API has extra functions for dumping and multiple deletion of services registered at the seap—server. FORTRAN bindings are not provided since seap will normally not be used directly by an application but implicitly by via visit.

6.2.1 Usage

To use the seap client functions, put one of the following lines of code in your program:

```plaintext
#include "seap.h" /* C */

use Seap; # Perl
```

In a Perl program, you have to create a seap—client object, that does nothing more than remembering the parameters that usually don’t change during a session. These are the hostname and port—number of the seap—server ($serverhost, $serverport) and parameters that specify how often ($maxpoll) a client polls the seap—server for information and how long he will wait between two queries ($pollinterval).

```plaintext
$seap = new Seap( $pollinterval, $maxpoll );
```
6.2.2 seap_publish

```c
int seap_publish(const char *service, const char *passwd, const char *host, int port);
```

```php
-ok = $seap->publish( $service, $passwd, $host, $port);
```

**Description:**
register a service with the 'seap_server'.

**Parameters:**
- `service` the name of the service to publish.
- `passwd` the (secret) passwd associated with the service.
- `host` the name of the host that provides the service if '*' is used, the seap_server replaces this by the name of the host that called 'seap_publish'.
- `port` the portnumber where the serving host listens for clients.

**Return Values:**
The function returns 1 on success, 0 otherwise.

---

6.2.3 seap_unpublish

```c
int seap_unpublish(const char *service, const char *passwd);
```

```php
-ok = $seap->unpublish( $service, $passwd);
```

**Description:**
delete a previously registered service on the 'seap_server'.

**Parameters:**
- `service` the name of the service to delete.
- `passwd` the (secret) passwd associated with the service.

**Return Values:**
The function returns 1 on success, 0 otherwise.

---

6.2.4 seap_query

```c
int seap_query(const char *service, const char *passwd, char * const host, int * const port, int pollinterval, int maxpoll);
```

```php
($host, $port) = $seap->query( $service, $passwd );
```

**Description:**
queries the 'seap_server' for a named service.
Parameters:

- **service**: the name of the service to query.
- **passwd**: the (secret) passwd associated with the service.
- **host,port**: name and port-number of the server (output parameters!).
- **pollinterval**: query the server every pollinterval seconds if the service is not registered at the 'seap server'.
- **maxpoll**: don’t query more than maxpoll times.

In the Perl binding pollinterval and maxpoll are taken from the seap–object.

Return Values:

The function returns 1 on success, 0 otherwise.

### 6.3 seap demo clients

The Tk-client seap is more or less self-explanatory. Per default, it queries the seap server every 5 seconds for all registered services and displays them. Using the File-Menu, you can switch of this auto-update feature (but not change the interval). You may also manually update the information (Update-Button), delete an entry (by selecting it and pressing the 'Delete'-Button), modify it (by selecting it, pressing the 'Edit' Button, and changing it in the text input-field - with the Enter-key or the 'Insert:'-button the changes are accepted). You may also enter new entries in the text input-field, accepting the with the Enter-key or the 'Insert:'-button.

Seap is usually started with

```bash
seap -passwd=<passwd> [-height=<height>]
```

All of its actions only refer to services that have the passwd given on the command line. The optional height–parameter can be used to specify the number of lines in the display.

An alternate way of starting seap is

```bash
seap -master=<master_passwd> [-height=<height>]
```

With the correct master–passwd of the seap server, seap displays all published services. In master–mode it is possible to delete services with any passwd. However it is not possible to extract passwds or to enter or modify services. This keeps a certain amount of privacy for the users by not exposing their passwords to the maintainer of the seap server. For the seap server, only the combination of service-name and passwd needs to be unique. Therefore, in master–mode seap may display several services with the same service–name.

Sclient is a simple C–based seap–client. It can publish, query and unpublish services. The usage is:

```bash
sclient -p <service> <passwd> <host> <port> ; publish a service
sclient -q <service> <passwd> ; query a service
sclient -d <service> <passwd> ; unpublish a service
```
Chapter 7

Tools

7.1 seap – monitoring the seap–server

seap is a small Perl/Tk based monitoring tool for seap. In its normal mode, it displays a list of all services that are registered at the seap–server with a certain password. The GUI lets you unregister services, edit them or add new ones. The purpose of seap is to give you informations about your visit–servers and –clients. Typically, services will show up when your server is ready, disappear, when the client connects, and show up again when the client is disconnected. You may also use it to unregister old services that may remain registered when your visit–server application crashes.

Figure 7.1: seap displaying a couple of services related to the Game of life demo.

Usage:

seap [-height=<height> [-passwd=[$passwd]]]

The with optional height–parameter, you can choose the initial size of the window.

The optional verb+-passwd+–parameter can be used to specify the password on the commandline. Normally, you should not use this parameter, because other users can obtain your commandline with the ps–command. If you don’t specify a password, the GUI will prompt you for one.

seap also has a ’master–mode’. To operate in this mode, you have to know the master–password of the seap–server (as specified in .seapserverrc). When started in master–mode, all services are displayed, no matter which key is associated with them. Services can be unregistered, but you cannot edit them or add new ones. This restriction is not imposed by the tool but by the seap–server. The idea behind that is, that an administrator must be able to remove garbage from the seap–server but should not alter user data (see section 6).

Usage:

seap -master[=<master_passwd>] [-height=<height>]

Like in normal mode, you can specify the password on the commandline. You can switch between normal and master mode by selecting ’Password’ in the ’File’ menu. A dialog pops up that lets you change password and mode.
When `autoupdate` is active (the default), `seap` refreshes its list with current data from the `seap`–server every 5 seconds.

To unregister or edit a service, select it with the left mouse button. When an update occurs (either automatically or because you pressed the `update` button) before you have pressed the `delete` or `edit` button, your selection is canceled and you have to repeat it.

### 7.2 vbroker – attaching multiple visualizations

`vbroker` is a Perl/Tk based tool that lets you attach multiple `visit`–servers (aka visualizations) to a single `visit`–client (simulation). It does this by forwarding all send–requests from the client to all attached servers. Receive–requests however, are only forwarded to a single server. `vbroker` lets you choose at any time, which server will get receive–requests. This means that you may have multiple passive viewers but only one visualization may steer the application.

`vbroker` can also be used to record data from the simulation to one or more files, or replay previously recorded data.

The tool maintains a list of all requests of the simulation with informations about which visualizations received each request. While this is mainly intended for monitoring the status of the multiple visualizations, it can also be used to debug your application: If you put vbroker between your simulation and visualization, you get detailed information about the communication between them.

The main part of the GUI consists of three panels. Only one of then is visible at a time. With the 'Client connection / Simulation' panel you control the connection to the simulation, with the 'Server connections / Visualizations' panel you control the connections to the attached visualizations. The 'Messages' panel displays a history of the status messages that appear in the bottom line of the GUI.

The current version of VBroker may crash with a segmentation fault when you press the exit button. We are not sure whether this is caused by `visit` or by problems within Perl/Tk which are related to cleaning up fileevent–bindings at exit. Since this only occurs when you exit VBroker, we consider it a minor problem.

#### 7.2.1 The 'Client connection / Simulation’ panel

With this panel you control the connection to the simulation. This client can either be a simulation or a file.

![VBroker client and server panels in a typical Game of Life session as described in the example section 7.2.3.](image-url)
To connect to a simulation you first have to select 'Seap' and enter a service/password combination that the client simulation can connect to. Then press the 'Start' button. VBroker is now ready to connect to a simulation and the button text changes to 'Stop'. When a connection is established, the 'Pause' and 'Disconnect' buttons become active. With 'Pause/Cont' you can pause/continue responding to client requests without disconnecting. 'Disconnect' disconnects the client. When you press 'Stop' VBroker will disconnect the client and no longer announce its service or accept connections.

To read data from a file, select 'File' and enter the name of a file with pre-recorded data. Here, the 'Start/Stop' and 'Pause/Cont' buttons are used to start, stop and pause the replay. The 'file event dilatation' slider lets you control the speed of the replay. With a value of 1, data is replays with the same timing as it was recorded, larger values lead to slower, smaller values to faster replay. A value of 0 means replay as fast as possible.

In the lower part of the window all requests are listed as they are processed. The 'D' column shows the 'direction' (send or receive), 'id' is the request id, 'timestamp', 'vtype', and 'dim' are the corresponding parameters in the request envelope. In the 'servers' column, all servers are listed that process the request. Each server is identified by its service- or filename.

The entries in the 'D' and 'servers' columns are colored. Green means, that the request has been processed successfully, red means it has failed. Note that send requests even succeed when no server (visualization) is connected, because they are handled by VBroker. Receive requests on the other hand are passed to a 'master–server' (see next section). They can only succeed when such a master is active.

7.2.2 The 'Server connections / Visualizations' panel

This panel controls the connections to the servers (visualizations). The entry-fields at the top of the panel let you add servers. By giving their service/passwd combination you add 'real' servers. You can also add files, that record the requests from the client simulation. All servers are listed in the lower part of the panel. For each of them delete, pause/continue and disconnect/reconnect buttons are created that let you control their operation.

With the radio–button in the column named 'Master' one of the servers can be selected to be the 'master–server'. While all servers receive data from the client simulation, only the master gets the receive–requests. When no master is selected, receive requests from the client fail. If that happens, you will usually have to press the 'Disconnect' button in the 'Client connection / Simulation' panel to restart the connection.

With each server connection, the service/password or filename is colored on the screen to give you an impression about what is going on. A connected server is marked in green, a disconnected in red, a paused in yellow. Behind the name, the number of requests that have been processed by this server is printed.

7.2.3 Example session

This section demonstrates step by step how you can use VBroker to attach several visualizations to the game of life simulation that is described in section 8.2. We assume that you have installed visit with Perl–bindings and compiled the cgol.c located in <prefix>/demo/gol.

Step 1: connect a visualization to VBroker

Start the Perl/Tk gol visualization:

tkgol.pl -server cgol

You have to select a non–default service name here, because you don’t want to connect directly to the client.
Start VBroker (the program vbroker is installed in <prefix>\bin). Enter the 'Server connections / Visualizations' panel. Enter cgol1:dcgol as service/password combination and press the 'add' button at the right end of the line. In the lower part of the window a line is created for the connection, the name 'cgol1:dcgol' should be green to mark an active connection. Press on the diamond left to the name to make this the 'master–server'.

**Step 2:** connect to the simulation

Change to the 'Client connection / Simulation' panel. Enter 'cgol:dcgol' as the seap service/passwd combination. If not already active, click on the diamond at the left of the line to choose a seap–based connection (in contrast to a file connection). Press the 'Start' button. The server state should change from '-' to 'no conn'. This means that VBroker is waiting for a connection.

Start the simulation:

cgol -S 20 20 20 -i -g 10000

are reasonable parameters. It gives you a sufficiently large board, will calculate up to 10000 generations and insert a runner at a random position. You should immediately see the simulation and the visualization interact, as if they were connected directly. In the VBroker client panel, you can watch the requests being processed.

**Step 3:** connect a second visualization

Start another gol visualization:

tkgol.pl -server cgol2

Edit the service/password combination to read 'cgols:dcgol' and press 'add' again. The new visualization should now show the same data as the first one, but pressing 'Stop', 'Runner 1/2' or 'Flood' should have no effect on the simulation. By selecting the new visualization as the master, you can change that at any time.

**Step 4:** log data to a file

Enter 'cgol.log' as filename in the 'Server connections / Visualizations' panel and press 'add'. In addition to be displayed by the clients, all data is now logged in this file.

**Step 5:** replay the file

When you have recorded some data, press the 'Delete' button for 'cgol.log' to close the file. Switch to the 'Client connection / Simulation' panel and press 'Stop'. The cgol program will be disconnected from VBroker and cannot reconnect. Enter 'cgol.log' as filename, select the 'File'–diamond and press 'Start'. The contents of the file is now replayed and sent to the two visualizations. Note that you can alter the speed of the replay with the 'file event dilatation' slider.
Chapter 8

Demo Programs

This chapter gives a brief description of the programs located in the demo subdirectories. More information can be found in the source code. Here, we mainly describe the functionality and usage of the demos.

8.1 Test clients and servers

The programs listed in this section are located in the directory demo/test. During the installation only those demos that can be executed in your environment are installed to <prefix>/demo/test. The programs are just simple visit–clients and –servers that test the basic functionality of visit and the language bindings. Besides that, they do nothing useful. Especially the C versions vclient.c and vserv.c contain lots of comments in the source code.

8.1.1 vclient.c

A simple visit–client. Connects to a visit–server and sends and receives small amounts of data of all supported types. Service–name and key are hard–coded to vserv and demo_passwd. The received data is printed to the screen.

Usage:

vclient [-f <filename>] [-p] [-n <loops>]

If the -f option is given, all data that is sent to the visit–server will also be written into the specified file.
If the -p option is given, vclient polls the seap–server for the service until a connection to the visit–server is established.
if the -p option is given, the program cycles loops times through the send/receive calls before it exits.

8.1.2 vserv.c

A simple visit–server. Service–name and key are hard–coded to vserv and demo_passwd, so that vserv can co–operate with vclient. vserv accepts client connections in an infinite loop. All client requests are fulfilled. Data that is sent from the client is displayed (if the datatype is supported). If a client requests data, dummy values are generated and sent. If an unsupported datatype is requested, the client is disconnected from the server. vserv has no useful commandline parameters.
8.1.3 vclient.pl

A visit–client implemented in Perl. Although its functionality differs slightly from vclient.c it also co–operates with vserv.c. It connects to the server and sends and receives various data in normal and packed form and finally disconnects.

Note, that vclient.pl is not completely portable. It uses pack("I", ..) to create a packed array of INT32 values. This may fail on some platforms (see your perl documentation).

Usage:

vclient.pl [-service <service>] [-passwd <passwd>]
[-h <host>:<port>] [-f <filename>] [-S]

With -f given, all data that is sent to the server is also written to a file. Optionally, service name and key can be specified (if they differ from the default values). It is also possible to specify hostname and portnumber of the server directly. In that case, the seap–server is not queried. If -S is given, only send requests are issued, no data is received.

8.1.4 vserv.pl, tkserv.pl

Two demo visit–servers that have the same functionality as its C counterpart vserv.c. Only the output to the screen is slightly different. Like vserv.c, vserv.pl and tkserv.pl co–operate with vclient.c and vclient.pl and have no useful commandline parameters.

tkserv.pl demonstrates how a visit–server can be integrated in the a GUI based on Perl/Tk. The only trick is to bind appropriate 'fileevents' to the sockets of the visit–server.

8.1.5 VisitSimpleEg (AVS/Express)

This example is a counterpart to the (f)vclient.c demo program. It shows the functionality of the three visit macros visitserver, visitreader and visitwriter. AVS/Express network receives messages with different datatypes from the client program (Id=1) and sends messages of different datatypes (Id=2) to the client program.

The parameter of visitserver (SeapService, SeapPasswd and Interface) are changeable in the visitserver panel. The connection between AVS/Express and (f)vclient can be switched on and off during the run with the Listen toggle in this panel. For this the client program vclient should be started with the parameter -p and -n i. The status line in the visitserver panel show the actual state of the connection. The text color notifies the general state like a traffic light: green=connected, yellow=listening, red=not listening.

This example is part of the visit package of the IAC library and can be found in the Example folder of the IAC library section of the AVS/Express network editor.

8.1.6 fvclient.f

A demo visit–client implemented in Fortran. It has a similar functionality as its C counterpart and no commandline parameters.

8.1.7 sclient.c, querytime.c

sclient.c is a seap–client that can be used to publish, query, and unpublish services. It is described in detail in section 6.3.

querytime.c is a seap–client that displays the time needed to query a service from your seap–server. This program has no commandline parameters.
8.2 Game of Life

The example Game of life demonstrates how visit can be integrated in typical simulation program. The simulation 'cgol' is a C program that plays the well known 'Game of Life' in 3 spatial dimensions. This game simulates the evolution of a population on the basis of a few simple rules. The board is divided in cells which can either be populated or not. In each new generation, a cell survives if it has 5 or 6 neighbors. If an empty cell has 5 neighbors, a new inhabitant is born.

Usage:

```
cgol <options>
```

- `-S <x> <y> <z>`: size of 3D-Field
- `-g <maxgenerations>`: max. number of generations calculated
- `[-i]`: insert a runner type 1 at position 1,1,1
- `[-s <service>]`: contact point of the server (cgol)
- `[-p <passwd>]`: contact point of the server (dcgol)
- `[-v]`: verbose (off)

If the commandline option `-i` is given, the program inserts at runner (a constellation of living cells which walks through the 3d field) the beginning of the simulation. Further insertions of living cells can only be performed with a visualization/steering tool which is connected to cgol via visit.

After each generation (but at most once per second) the simulation program tries to connect to a visualization that has announced a service (default service name 'cgol', password 'dcgol', others can be specified on the command line) at the seap – server. If successful, cgol transfers the actual state of the 3d field at every generation. Additionally cgol requests a set of steering parameters from the visualization. Currently these parameters allow to stop or pause/continue the simulation and to insert runners (either running along on of the coordinate axis (type 1) or on a diagonal (type 2) or a randomly distributed population. If the insertion of a runner is requested, cgol asks for its position and orientation. If any action was requested, cgol sends an acknowledgment message to the visualization after the action is performed. This make sit easier for the visualization to reset its internal state.

We implemented two visualization tools to display and control cgol: tkgol.pl is a Perl/Tk script and shows a 2d projection of the 3d field; VisitGoLEg is a AVS/Express network which displays the field in 3d. Both examples include a Panel to control the simulation via visit.

8.2.1 VisitGoLEg (AVS/Express)

The VisitGoLEg example is part of the visit package of the IAC library and can be found in the Example folder of the IAC library section of the AVS/Express network editor. For the steering components of the visualization the visitserver panel has been extended. The control buttons and sliders are arranged below the control elements of the visit connection. So the user has only one panel to control the connection and the simulation.

The cgol part of the panel contains following elements:

**Pause**-button: When this checkbutton is activated the simulation stops its calculations and polls the visualization for the button state every 2 seconds. When the button is deactivated the simulation continues with the calculation.

**Stop**-button: When this checkbutton is activated the simulation terminates and the connection to Express will be closed.
Figure 8.1: Visualization and steering for the Game of Life simulation with AVS/Express.

Insert-button: The simulation inserts a runner at the position determined with the three sliders on the right side of the button. The runner will walk in the direction specified with the direction dial below the insert button. The value 1 corresponds to the x-axis, 2 to the y-axis and 3 to the z-axis. A negative value indicates that the runner walks in a negative direction. The value 0 is not allowed. When the simulation has inserted the runner, it will deactivate the button.

Random-button: If this checkbutton is activated a random number of cells in inner region of the field will be set. The slider besides the button describes how many of the inner cells should be set. When the simulation has inserted the cells, it will deactivate the button.

Figure 8.2 shows the top-level network for this example. For the GUI and the communication there are two macros (GUI) which contain the corresponding macros and modules. With this abstraction the data flow between the component is identifiable. The output port of the communication macro contains the actual 3d data field which will be delivered to the visualization macros Axis3D, Bounds and Glyph. There are several connections between the GUI and the communication macro which are responsible for the steering parameters (state of the checkbuttons) and the insert positions.

The next figure 8.3 shows the contents of the communication macro. There are two visitwriter modules and three visitreader modules. The first reader on the right side reads a string from the visit connection, which contains status messages of cgol. The second reader gets the acknowledgment values, which will be used for resetting the action button when the corresponding action is performed. The last reader is responsible for the 3d field, which has the datatype INT32. The three output ports n1, n2, n3 of the reader are concatenated to a array of dimensions which are together with the integer data stream of field contents (DataInt port of visitreader) a input port of the module uniform_scalar_field. The first writer (left side of the network) sends the insert position back to the
simulation (if requested), the second is responsible for the steering parameters.

Figure 8.3: Network for the communication between AVS/Express and cgol.
8.2.2 tkgol.pl

tkgol.pl is a simple but non-trivial example of a visit-server embedded in a Perl/Tk script. It displays the results of the cgol simulation and supports almost all of its steering capabilities. tkgol.pl has buttons to pause, continue, and stop the simulation as well as buttons to insert runners of type 1 or 2 or a random population (the ‘Flood’ button). The only limitation is that it is not possible to enter the position and orientation of the runners or the size of the random population. For these parameters, random values are sent to the simulation.

Usage:

```
tkgol.pl <options>
```

[-service <service>] contact point of the server (cgol)
[-passwd <passwd>] contact point of the server (dgcgol)
[-verbose] verbose (off)

Figure 8.4: Visualize and steer the Game of Life simulation with Perl/Tk
Chapter 9

Installation and Porting

For installation please also read the file INSTALL. It may be more up to date.

9.1 Prerequisites

We have tested visit on a couple of UNIX-platforms including Solaris 2.6, Linux 2.2, AIX 4.3, IRIX 5.6, and Unicos/mk. We expect it to work with only minor modifications on most UNIX platforms with IEEE arithmetics. To install visit you need at least an ANSI C compiler and a make utility. Since the seap–server which is a necessary part of visit is implemented in Perl 5, you need access to at least one machine with Perl 5. Some of the included tools and examples will also use Perl 5.005 with Perl/Tk 800, others use C++ or AVS/Express. However, these tools are not required for using visit.

9.2 Quick Installation

For a first shot, try the following. Unpack the distribution in a directory you have write access to:

```
gunzip -c visit-1.0.tgz | tar xf -
```

Change directory to the just unpacked source distribution and run configure:

```
cd visit-1.0
./configure --prefix=/tmp/visit
```

With the prefix parameter you may specify the installation directory. We suggest, that you use a directory which is exclusively for visit, because visit will spread into a couple of subdirectories during installation and you will have less trouble removing or updating visit if it does not mix with other software. If you omit --prefix visit will be installed in /usr/local/visit.

Build and install visit:

```
make
make install
```

Don’t try to just type make install, the perl bindings will not be generated properly!

visit and seap are now installed. Before you can use them, you need to configure and start a seap–server. Decide, on which machine you want to run the seap–server. On this machine, you need a working Perl 5 installation. For our tests we have used perl 5.005_02, but we expect the server to work with any Perl 5.x. For testing, you may use your own userid on your workstation, on the long term we suggest to use a separate account on a machine with high availability.
On the chosen machine, create a config-file for the seap-server. The syntax of the config-file is described in more detail in 6.1. The file basically looks like this:

```plaintext
seap_port : 4711 # an arbitrary port > 1000
seap_pidfile : /tmp/seap_srv.pid # remembers the PID
seap_passwd : my_passwd # the master password
seap_debug : 0 # 1 for extensive stdout-logging
```

Start the seap-server:

```
seap_server -f <config-file>
```

where `<config-file>` is the full path of your config-file. The default location of the config-file is `$HOME/.seapserverrc`. If you place it there, you may omit the `-f`-parameter.

In your visit-installation, edit the file `<prefix>/etc/visitrc`, where `prefix` is the parameter you gave to configure.

```plaintext
seap_server : seapsrv.mydomain.com # your seap_server
seap_port : 4711 # port of seap_server
```

These are the system-defaults. If a file `.visitrc` exists in a users home-directory, values therein override those in `<prefix>/etc/visitrc`.

At this time the installation of visit is completed. You should now compile and run the demo programs in order to verify that your installation was successful.

### 9.3 Test the installation

Before you can test visit, you have to complete the installation (see previous section), because the demos expect the include-files and libraries to be in the proper locations in `<prefix>`.

Change directory to `demo/test`. All of the test programs and Makefiles are located there. Start with testing your seap installation. Build the seap demo-client `sclient`:

```
make sclient
```

Register a service at the seap-server:

```
./sclient -p test_service test_key test_host 99
```

where `test_service`, `test_key`, `test_host` can be any strings and `99` any number. If the program finishes quietly, everything is ok. If you get something like

```
_seap_get_server: failed to read 'seap_server' from rc-file
```

your visitrc file has a wrong syntax or could not be found. If you get something like

```
seap_publish connect to 'seapsrv:4711': Connection refused
```

either your seap-server is not running or the entries in your visitrc file are wrong.

Query the just registered service:

```
./sclient -q test_service test_key
```

The answer should be:
host = 'test_host', port = 99

Unregister the service:

./sclient -d test_service test_key

The program should finish quietly. If you query the service again, the answer should be

query failed

See section 6.3 for a detailed description of sclient. If everything looks fine until here, your seap–installation seems to work properly and you may start to test visit with a trivial pair of client and server:

make vserv vclient

First start vserv and then vclient in a different window (or fvclient, if you have a Fortran compiler) and watch them exchange data. Both programs will print information messages on the screen. The client should exit after a few messages. The server will complain about a broken pipe and then wait for a new connection. You may start the client again to repeat that. If the server is running on a machine that does not support all the data types sent by the client, it will complain and disconnect the client.

For this example to work, the seap_server must be running. If you have installed the perl bindings (default, if you have perl5) you can substitute either vserv or vclient or both with vserv.pl or vclient.pl. To use the perl scripts, you have to set the environment variable PERL5LIB so that perl can find the visit modules. You may source the script visit_perl5lib to do so:

. <prefix>/bin/visit_perl5lib

where <prefix> is the the value specified to configure. If anything goes wrong, take a look at the next section. Configure has a couple of optional parameters that may help. For more tests, look at the other demo programs, which are described in chapter 8.

## 9.4 Configure options

Configure tries to guess what your system looks like, which compilers and tools are present and from that information creates Makefiles and other files needed for proper compilation and installation of visit. Configure is completely non–interactive, but has a couple of command–line options, that influence its behaviour. Also certain variables will also be used if set.

visit contains certain optional parts and features. By default, configure will build everything that it believes is possible on your system. E.g. it will build Fortran–bindings, when it finds a Fortran compiler. What follows is a list of parameters and variables (with the default value in braces) and a short description of the way they work.

```
--prefix=<prefix> [ /usr/local/visit ]
the top–level installation directory for visit.
--with-perl=<yes|no|visit|perl>
--with-perl ( same as yes )
--without-perl ( same as no )
```

Configures the perl–bindings of visit. If not given, the perl–bindings are build, if and only if your PATH contains a perl (version 5.x) interpreter. If you select no, no perl–bindungs are build.
By default, or when you select either yes or visit the perl modules will be installed in <prefix>/lib/perl5/... In that case users must set the environment variable PERL5LIB in order to use them. For that purpose a script named visit_perl5lib is automagically created and installed in <prefix>/bin. Users have to source that script. Too avoid this, you may select --with-perl=perl. In that case, the modules will be installed in the same directories perl uses. Configure will stop with an error message, if you request the perl–bindings to be build and perl is not available.

--with-seapperl=<yes|no|visit|perl>
--without-seapperl ( same as yes )

Configures seap. If not given, the seap–server and the perl–bindings for seap will be build, if and only if a perl (version 5.x) interpreter is found. This is separated from the perl–bindings of visit, because you need a seap–server on at least one machine, but can do without visit’s perl–bindings. The installation directories are selected as with --with-perl. If you don’t specify it explicitly (with perl or visit) it will use the same as given for --with-perl, (and vice versa).

--with-copt=<options> [ -O]
options, that are passed to the C compiler. As the name suggests, it is intended for optimization options, but can of course be used for other type of options, too.

--with-fopt=<options> [ -O]
options, that are passed to the Fortran compiler. As the name suggests, it is intended for optimization options, but can of course be used for other type of options, too.

--with-debug
If given, all sources will be compiled with -DDEBUG, which will lead to exhaustive debugging output. Additionally, the compiler options are changed to -g.

--with-swig
This option is for porters/developers only. Swig is a tool that supports the creation modules for perl and other scripting languages like tcl and python. You only need it, if you want to change the core of the perl–bindings (the file visit_perl/VisitRaw.i).

--with-blocking
The ability of visit to time out TCP/IP communication partners that don’t respond or respond to slowly relies on non–blocking sockets. Therefore sockets are opened non–blocking per default. On the Cray T3E, we experienced infrequent crashes of non–blocking visit–sockets in Fortran. If you want to use the Fortran bindings on the T3E or experience similar problems on other platforms, you may specify the option --with-blocking. It compiles the visit client functions to use blocking sockets. This means that your simulation will never time out a visualization that hangs. However, if the simulation terminates or crashes, your simulation will shut down the connection properly.

--help
The configure–script itself is created by the gnu–tool autoconf from a file named configure.in. Besides the options listed above there are a couple of other generic options, which you might find more or less useful. With this option you get a comprehensive list of them.

CC, F77, PERL5, SWIG, CPP, INSTALL
Explicitly set the compilers and tools you want to use. This is particularly useful if you have several C or Fortran compilers installed and don’t like the one that configure selects by default, e.g:
There are lots of other variables (like CFLAGS, FFLAGS, LIBS) that might be useful. Check the autoconf documentation or look into the configure script.

9.5 Porting hints

The configure script that is part of the visit—distribution tries to detect a couple of things that may differ between platforms and modify Makefiles and part of the sources to overcome these differences. The current script only looks at things that are relevant for machines that we have access to. If you have other platforms, it may be necessary to add new tests. The configure script is automatically generated by the GNU autoconf utility, so please edit configure.in if required. The next two sections treat the problems of Fortran support and data representations which will most probably be an issue. On more exotic UNIX variants, you may also experience problems with socket options and header—files.

If you experience any problems we would like to hear from you, not matter whether you have solved it yourself or not.

9.5.1 Fortran issues

Unfortunately, there is no standard for mixed language programming between C and Fortran 77. Most of the problems arise with the naming conventions of symbols and the passing of string parameters to a Fortran subroutine. While C functions generate a symbol of the same name a Fortran function MYDEAR may appear as mydear, MYDEAR _mydear, MYDEAR_, ... or whatever you can think of. This may even vary between Fortran compilers on the same machine. The configure script tests for a couple of common cases and modifies the file visitf.c which contains the Fortran bindings accordingly. Be aware, that this means that if you have configured visit with a certain Fortran compiler, the bindings may not work with an other compiler on the same machine! If your compiler generates symbols that are not recognized by configure you may be forced to extend the test and edit visitf.c.

Another typical problem is the handling of function parameters of type CHARACTER. Many compilers implicitly add the declared length of the variable to the parameter list. On SGI/CRAY systems a CHARACTER variable is represented by a structure named _fcd. Configure only supports these two cases.

9.5.2 Data types on new platforms

On platforms with IEEE arithmetics, the representation of numerical data types only differs by size and byte order. The latter case is handled by visit at runtime. For the sizes, visit provides only support for integer types. visit uses 16 and 32 bit integers and configure typedefs the appropriate C types to vint16 and vint32. The macro HAS_VINT16 is defined if and only if a 16 bit integer type exists. The current version of visit generally assumes that a double (and a Fortran DOUBLE PRECISION) have 64 bits. This is hard—coded at several places and will be changed in a later release.

9.5.3 Defining new data types

visit supports only a limited number of data types (see 2.5) and has no mechanism for defining new types on the application level. However, it is quite simple to define new types by modifying the library itself. In total, there are five files to modify (or three, if don’t need the perl—bindings):
1. define it in visit.h
2. modify the visit_sizeof function in visit.c to return the correct size of the new datatype.
3. modify the _visit_srv_convert function in visit_srv.c.
4. modify the functions _name2vtype and _vtype2name in Visit.pm.
5. define it in VisitRaw.i and modify the functions _visit_AV2data and _visit_data2AV in the same file.

E.g. assume, you want to add a 32–bit float value. Then you would do the following:

1. add a line:

```c
#define VISIT_FLOAT32 6 /* any unused positive value */
```
to visit.h

2. edit the function visit_sizeof in visit.c by adding an extra case to the switch statement. This should set size to the number of bytes of the new datatype.

```c
case VISIT_FLOAT32:
    size = 4;
    break;
```

3. edit the function _visit_srv_convert function in visit_srv.c by adding a case to the switch statement. This should define the conversion that is required between machines of different endianess for the new datatype.

```c
case VISIT_FLOAT32:
    ctoh32arr(data, size / visit_sizeof(VISIT_FLOAT32),
               aflag);
    break;
```

4. edit the file Visit.pm. In the function _name2vtype add a line:

```perl
$vtype = $VisitRaw::VISIT_FLOAT32 if( $name eq 'FLOAT32' );
```

In the function _vtype2name add a line:

```perl
$name = 'FLOAT32' if( $vtype == $VisitRaw::VISIT_FLOAT32 );
```

These functions provide conversions between the integer representation of the data type in the C implementation and the string representation used by the perl–bindings.

5. repeat the define–directive from visit.h in VisitRaw.i

```c
#define VISIT_FLOAT32 6 /* same value as in visit.h */
```

and modify the functions _visit_AV2data and _visit_data2AV. These functions convert Perl arrays of the data type to C arrays and back. In _visit_AV2data you would add a new case to the switch on vtype:

```c
    case VISIT_FLOAT32:
    {
        float *fp = Cdata;
        for(i=0; i<n; i++) {
            tv = av_fetch(AVdata, i, 0);
            fp[i] = SvNV(*tv);
        }
    }
    break;
```

In _visit_data2AV it would be:
9.5. PORTING HINTS

```c
case VISIT_FLOAT32:
{
  float *fp = Cdata;
  
  for(i=0; i<n; i++) {
    svs[i] = sv_newmortal();
    sv_setnv(svs[i], fp[i]);
  }
}
break;
```

Note that this assumes that `sizeof(float)` is 4 on all of your platforms – like the current implementation assumes that `sizeof(double)` is always 8.