

Submitted to: International Computational Accelerator Physics Conference ICAP 2000
Darmstadt, Germany, 9/11-14, 2000

Simlink: a software substrate for physical simulations

K.A. Brown

Brookhaven National Laboratory, Upton, New York 11973-5000

Tzi-cker Chiueh

Experimental Computer Systems Laboratory, Computer Science Department, SUNY Stony Brook, Stony Brook, NY
11794-4400

The simlink server is an HTTP/1.1 server which doesn't serve html documents, but instead, serves the output of a physical simulator. A host system user defines what simulator will be run by the server through a short simple configuration file, which can be specified at invocation time. Since the server uses http protocol it is capable of sending data out to http clients. A library is provided with the distribution which allows new clients to be constructed using the simlink API. This paper will describe the system and its design and demonstrate how it is used with BNL's Collider Accelerator online modeling environment.

I. INTRODUCTION

The goal of the simlink project is to build a middleware substrate that transforms simulators into servers, giving them network communication and interprocess communication capabilities. This will allow us to dynamically link these applications to other applications. For example, this would provide trusted high quality models to real-time controls systems. In many cases the simulators may be legacy applications which are well trusted and relatively free of bugs, but do not have the capabilities of interprocess or network communication. One option would be to modify the simulators to give them these abilities. This may be done if one has access to the source code, but it then requires modifying all the simulators, and introducing new bugs and possibly compromising the degree of trust in the simulator. Another approach is to have another application which connects to the simulators, which sends and intercepts data to and from the simulator. There are various ways to do this. One could link directly to the simulators address space, in much the same way as a debugger. A simpler approach is to just redirect the simulators input/output (I/O) stream. This is the approach taken in the simlink project. Simlink is basically an I/O redirection substrate which allows interprocess/internet communication to take place between applications that otherwise wouldn't have such capabilities.

II. MOTIVATION

The RHIC/AGS online modeling environment, developed at Brookhaven National Laboratory (BNL) by sci-

entists in the the Collider Accelerator Department (C-AD), is designed to interface high quality, trusted simulations of the accelerators to the controls system running the accelerators [1,2]. There are many components which comprise the Collider Accelerators. There are four alternating gradient synchrotrons which have a range in age of almost 40 years, from the Alternating Gradient Synchrotron (AGS), built in 1959, to the Relativistic Heavy Ion Collider (RHIC), which just became operational. The simlink system is designed to allow different simulation packages (particularly legacy applications) to be interfaced uniformly to the online modeling environment.

III. RELATED WORK

A primary goal of the RHIC/AGS online modeling environment is to provide realistic models for automated beam control and shaping systems [3]. Such systems are necessarily complex, since they involve a union of controls systems, instrumentation, and beam physics. From the computer science perspective these become issues of uniform interfaces and protocols, database management, and even system architecture. Large, complex accelerators such as RHIC truly require many functions to be automated [1,2].

One critical component to building automated control at the C-AD complex is the cdev C++ library [4]. Cdev (common device) provides a standard interface between an application and one or more underlying control packages or systems. cdev is widely used at many particle accelerator institutions, including BNL. The current model server at BNL interfaces to the controls system through the cdev interface and is built as a cdev generic server. The simlink server will be used at BNL to connect to the cdev model server and bring simulations into the system not previously reachable through the existing online modeling interface.

IV. SYSTEM OVERVIEW

Simlink is designed to be linked to a single simulator package, as specified in a very simple configuration file. Command line options at invocation allow specifying a configuration file and a socket port to allow client connections.

The server design was modeled on the device driver paradigm, with a top part, the request/response handler and a bottom part, the simulation handler. Figure 1 shows a simplified diagram of these basic blocks. Data from the simulator is put into a buffer in the simulation handler and transferred to the request/response handler when requested. A simulator instance is invoked via a request from the request/response handler. All messages and data that pass between the request/response handler and simulation handler are mediated by a single negotiator. Communication between the simulation handler and the simulator is specified in the configuration file and is either through unix sockets or named pipes. The request/response handler communicates with the negotiator (of which it is a child) through shared memory using semaphore synchronizations.

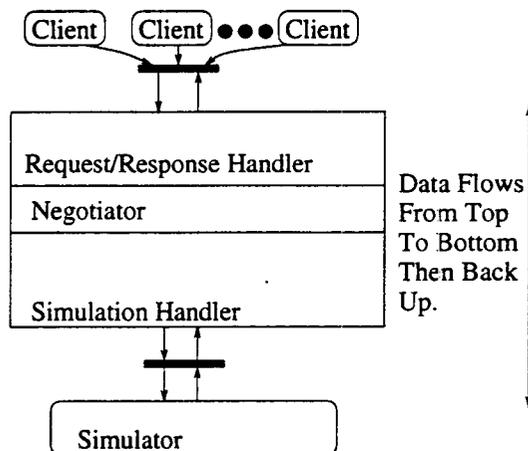


FIG. 1. Simlinks basic blocks

Simlink uses a limited http/1.1 protocol for the client side interface. The full protocol is accepted, but not fully implemented. Only the subset of the protocol required to trigger the simulator to run and to place a valid header on the response is implemented. The ability to send data from the server to the simulator is in place. This allows custom built clients to send more complicated requests with data.

Simlink includes, both in design and to a limited extent in the current implementation, various degrees of fault tolerance. Fault tolerance can be divided into three parts: error detection, classification, and actions. For error detection every class in the system includes a private data member error code and a public member function which returns the error code. A single enumeration type has been built which includes all errors (all error codes in all the classes are of this type). The errors will be used for error logging and for fault detection. In the fault detection design each class object will have a corresponding shared memory variable, which will contain the error code for that class. The total number of error codes in shared memory is relatively small, just the same as the number of class objects in the server (about

25 variables). A separate process (forked from the main server at start-up time) can monitor these variables and also periodically send test messages to the server. Classification of the errors will be done in a special class in the testing process. The actions will be classified based on the error classifications and will vary from deleting a faulty class (and re-initializing) to stopping and restarting the server. Logging of such events is critical to fixing bugs, and is an integral part of the error system.

In the next version the system will be made more versatile by allowing multiple request/response handlers, and perhaps even multiple negotiators and simulation handlers, to allow centralized but highly flexible selection of simulators. The apache web server uses a similar system of "virtual" connections at its top level interface [5]. The infrastructure for such a multiplex/demultiplex design is already in place in the simlink architecture. The synchronization between the multiple handlers will be implemented through test variables in shared memory. This mechanism is already in place and being used to synchronize between the request handler and the response handler. Expanding this to allow synchronization between multiple request/response handlers is therefore not difficult. The Apache server employs a similar mechanism, using what they call a scoreboard in shared memory.

The entire system was written in C++. Currently it runs only on the SGI platform (IRIX 6.3), and porting over to Linux is almost complete. We also plan to port over to Solaris 2.5.1. The system, whenever possible, uses POSIX standards, with the exception of the shared memory interface, which is System V.

V. SYSTEM DESIGN

Figure 2 shows the data flow model for the system, showing the interaction between the request/response handler, the negotiator, and simulation handler (which is broken up into a simulation handler and a data handler). The dashed lines in the diagram demarcate the different processes. A boxed rectangle on top of the dashed line represents a communication port. The main server consists of the negotiator and the simulation/data handlers. The request/response handler is a child process of the main server. Simulator instances currently are short lived child processes of the main server. The module buffer ram is a random access memory module built into the server.

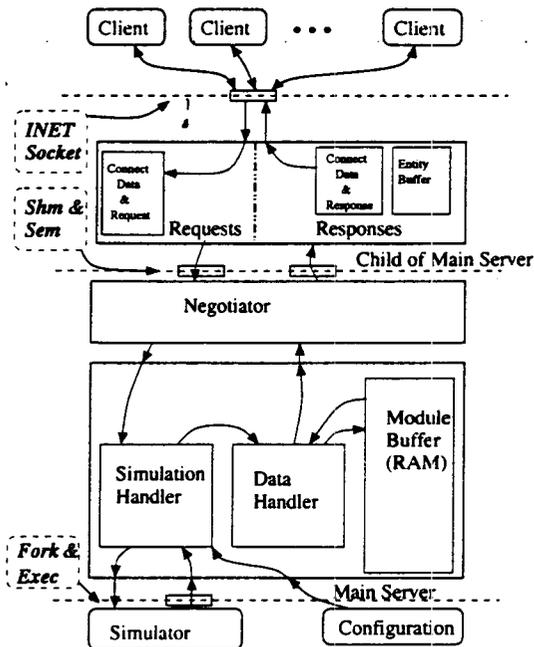


FIG. 2. Simlink data flow model

A. Simulation Handler

The simulation handler keeps a private data structure containing the information from the configuration file. This data structure can be modified through a function call, allowing dynamic change in simulator configuration. The simulator specified in the configuration file is executed as a child process. The data is then read in through either a named pipe or a unix socket and the data handler puts it into the buffer ram.

Since multiple forms of communication are being used, we developed an I/O class that creates a uniform interface to the I/O subsystems. This I/O class then interfaces with a lower-level class for the actual interprocess communication.

The higher-level I/O class is intended to be used by clients to allow a simple and uniform interface to sockets and pipes. A private member, $uici^* ps$, is a pointer to the lower-level class $uici$. The $uici$ class is a C++ version of the $uici$ system described in [Robbins [6]]. We expanded and generalized this class using the sockets interfaces described in [Chan [7]] and in [Stevens [9,10]].

Since the I/O interface to the simulator will not always be through a single file descriptor, we put the I/O objects into lists. The simulation handler has a pointer to the simulation class, which contains a list of unix socket I/O objects and a list of fifo I/O objects. Through pointers to these list objects we are able to control which file descriptors to use for input and output. All of these connections are defined in the configuration file and kept in data structures in the simulation handler.

B. Negotiator

The negotiator is encapsulated into a single class containing pointers to the simulation handler class, to the request handler class, and to the response handler class. There is only a single nontrivial member function to this class, called `pazu_negotiator()`. From within this function we make the request/response handler a child process, we setup the shared memory interface for communication to the request/response handler, and we instantiate all the simulation handler and data handler classes. The negotiator waits (blocking through semaphores) for a request to be sent from the request/response handler. It then sends the request down to the simulation handler and waits for it to complete. The data is retrieved from the data handler and sent back up to the response handler (which has been waiting for the reply back, again by sitting on a semaphore). The negotiator then goes back and waits for another request. Figure 3 shows a representation of this process.

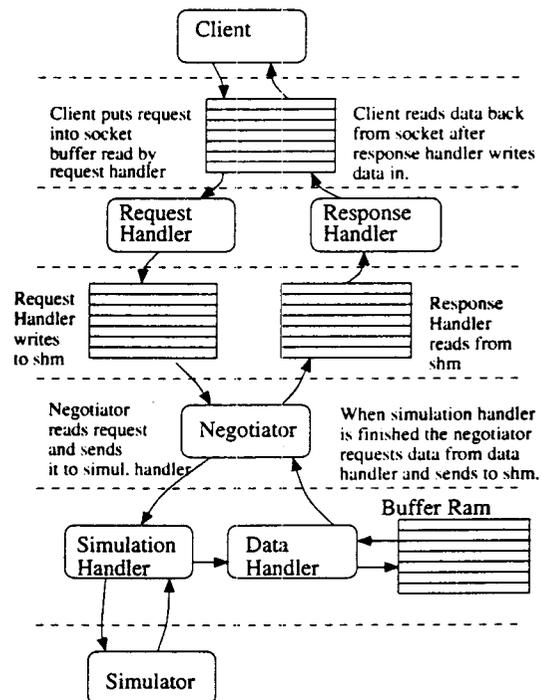


FIG. 3. Request/Response handling process

C. Request/Response Handler

The request handler contains pointers to a request record class, which contains a pointer to a client connection class. These two classes take care of all the request handling, including the parsing of the http request header and keeping track of the socket connection data. For the current version, this implementation is more complicated than really needed, but it contains the infrastructure for expanding to multiple request/response handlers.

The algorithm for taking a request and then sending it back out to the client relies on using semaphores and shared memory variables to ensure the state of the request is correct for the given operation [see Chan [7]]. Although such a mechanism is not required, it is an important building block to the more complicated system of multiple request handlers, in which synchronization is required.

We ensure secure connections, as much as possible, by not blocking on the request or response I/O. Non-blocking I/O is also an important building block to having a true multiplex/de-multiplex design [see Stevens [9,10]].

D. Simlink Protocol

The interface to clients uses the HTTP/1.1 protocol. This protocol is very well defined and contains almost all the functionality we require. There is a small subset of calls that we have added (calling this new protocol SLTP/1.0). The complete HTTP/1.1 protocol is fully described at the w3.org website [8]. The sltp-specific headers are defined but not completely implemented. These are only intended to be used by non-web browser clients, written specifically for communication with simlink.

SLTP specific header calls:

1. UseModel: allows specifying which simulator to connect to (for the next version, which will allow multiple simulators)
2. GetData: In some instances only a specific subset of the data is needed. This will allow asking for a specific range.
3. HostUser & HostUserPID: For future authentication. (encrypted passwords could be attached.)
4. HostReturnPort: To allow connectionless communication (client sends a request and then disconnects to do other work. Then the server connects back to client, using this port.)

When a client connection is made the message is sent to a parser, which is a member function of the request record class. This parser follows the syntax and grammar as described in [8] (note: we do not parse the full protocol, just the part we require, as noted in section IV). The following code shows a small segment of this parsing. A token number is taken from a set of enumeration types, listing the full http/1.1 protocol, by comparing the various strings in the message.

```
int
SL_Read_RQ::parserequest()
{
    int tokennum, num;
    char *lineptr, *tokenptr;
    char delim[]="\012\014\015\040\t,;()";
```

```
char delim[]="\012\014\015\040\t,;()";
char *colptr=NULL;
int hmlength=0;

lineptr = raw_request;
num=0;
for(tokennum = 1;
    (tokenptr = strtok(lineptr, delim));
    lineptr = NULL, tokennum++)
{
    if((num=slp->ismethod(tokenptr))>0)
        switch(num)
        {
            case 1: // GET
                if(method != NULL) delete[] method;
                method = new char[strlen(tokenptr)+1];
                strcpy(method, tokenptr);
                method_index=num;
            // request uri
            if((tokenptr = strtok(NULL, delim))!=NULL)
            {
                if(uri != NULL) delete[] uri;
                uri = new char[strlen(tokenptr)+1];
                strcpy(uri, tokenptr);
            }
            // the http/sltp version
            if((tokenptr = strtok(NULL, delim))!=NULL)
            {
                if(protocol != NULL) delete[] protocol;
                protocol = new char[strlen(tokenptr)+1];
                strcpy(protocol, tokenptr);
            }
            ...
            // parse out whether HTTP or SLTP
        }
        break;
        case 2: // HEAD
            ...
```

We test whether the client sent an HTTP message or an SLTP message. For our simple grammars this parser works well.

For the response message we again enumerate all the HTTP status codes and send out the appropriate message. Currently we send back only OK or BAD_REQUEST.

E. Fault Tolerance

The fault detection and correction system is currently incomplete, though the infrastructure is in place. The main component of the system is the error code enumeration type.

```
enum P_Error{
    PAZU_OK,
    //=== fatal errors
    // PAZU BOTTOM PART
    PAZU_,
    PAZU_RAMREADEROR,
```

```

PAZU_SIMUL_,
PAZU_SIMUL_FORK,
PAZU_SIMUL_EXEC,
PAZU_DATA_,
PIOLIST_,
P_SIMUL_,
P_SIMUL_READCONFIG,
P_SIMUL_MAXARGS,
P_SIMUL_MAXFILES,
...
};

```

Every class contains a private member of this type. For example, in the main server class there are the members `pazu_error` and `get_error()`.

```

class SL_Server{
private:
// where a well known port resides,
// actually does nothing
P_IO *main_server;
int ms_port;
char server_host[MAXHOSTNAMELEN+1];
char *server_name;
int hostid;
pid_t server_parent;
pid_t server_pid;
uid_t server_uid;
gid_t server_gid;
int sports[MaxPort];
int chunksize;
// open fd of main server
int msfd;

P_Error pazu_error;

public:
...
P_Error get_error(){return pazu_error;};
...
};

```

In the constructor of each class the `pazu_error` variable is set to be `PAZU_OK`. Within the member functions this variable can be changed by non-normal events.

F. Outstanding Issues

The following is a list of known bugs and deficiencies.

1. No error logging in place yet.
2. Buffer RAM is not dynamically expandable. This just needs programming time to fix it up so that the number of ram cells can be dynamically managed.
3. All request messages are accepted but only a minimal amount of sanity checking is performed on incoming messages.

The following is a list of upgrades to be included in the next version.

1. Fault tolerance as described in this report will be fully implemented.
2. Multiple request/response handlers will be included. This will allow creation of multiple socket ports to direct messages to and relieve the bottleneck of a single port.
3. The complete protocol as described in section [V D] will be implemented.

VI. USER INTERFACES

There are three user interfaces. First we describe the syntax of the configuration file. Second, the server takes a small set of command line arguments. Finally we will show a simple example client.

A. Configuration File Syntax

All lines beginning with “!” are ignored by the server. As a bare minimum the user must supply a `pname`. Redundant items are ignored. Keywords are: `pname`, `ppath`, `parg`, `dfile`, and `protocol`.

Configuration syntax:

1. `pname` filename
filename is the program executable and path
2. `ppath` filepath
filepath is the path to any program files
the communication protocol used for these files is define by *protocol* (see below)
3. `parg` type switch arg
switch and *arg* are the *pname* executable arguments
types are:
 - in = input
 - out = output
 - sw = switch
 - var = variable set
if *switch* is set to “null”, then this is a non-switch argument (e.g., `runprog inputfile` rather than `runprog -sw inputfile`)
4. `dfile` type filename protocol
standard files used by *pname* executable
types are in and out
protocol can be file, fifo, socket:
 - if file, simulator creates it, we ignore it
 - if fifo, we control the pipe
 - if socket, we control the pipe, opened as a UNIX Socket
5. `protocol` ioprotoocol
i/o protocol link for executable
choices are fifo, socket, or file

Here are two examples. In the first example the files DUMP and ECHO are set to be ignored. They don't need to be in the configuration file, we just show them in this example for illustration purposes. The program takes an argument, which is just the name of a unix socket.

```
pname sock_cls
ppath /usr/people/kbrown/laputa/bin
parg out null testserv
dfile out DUMP file
dfile out ECHO file
protocol socket
```

At the command line one might invoke this as:
%/usr/people/kbrown/laputa/bin/sock_cls testserv

In the next example pazu.print is specified twice. The server ignores the second one.

```
pname bnlmad
ppath /usr/people/kbrown/bin/
parg in -da agsmad.config
parg out -pr pazu.print
parg out -pr pazu.print
dfile out TWISS fifo
dfile out ECHO file
dfile out DUMP file
protocol file
```

At the command line one might invoke this as:
%/usr/people/kbrown/bin/bnlmad -da agsmad.config -pr pazu.print

B. Server Invocation

The server can be invoked with no arguments. In this case the default socket port used is 9669 and the default configuration file used is pazu.config.

For example, to show the version and a short help message:

```
261% simlink -v -h
simlink v.0.1 SLTP/0.9
Usage: simlink[-sp number][--scf file][-h][-v]
-sp number | --socketport number
        define a new default socket port number.

--scf file | --simconfigfile file
        define a new simulator configuration file.

-h | --help
        this message.

-v | --version
        version information.
```

To stop the server a "kill -INT" signal can be sent. The server has a signal handler which will catch the interrupt and exit gracefully. This includes all deallocation of data structures, killing child processes, and detaching

from share memory and semaphores. It is safe to stop the process with an interrupt, since care was taken to block interrupts during critical sections of the programs execution (for example, when accessing share memory or semaphores).

For example:

```
// Note: signals are blocked while
// accessing shared memory.
if(rq_mess->shm_ok()) {
    sigprocmask(SIG_BLOCK,&negsigset,NULL);
    rq_mess->receive((void*)clientname,&bsize,
                    &communicatefd,
                    (void*)currentreq,
                    &hdsiz,
                    SERVER_RECV, &rmtype);
    currentreq->connection->server=sls->ms_pointer();
    sigprocmask(SIG_UNBLOCK,&negsigset,NULL);

    slr->setclient(clientname, bsize, communicatefd);
```

C. Client API

The following shows a simple client program, using our I/O class for communication. The data sent back from the server is then written to stdout. This should be linked to the libpazu.a library, which is part of the distribution. A client doesn't have to use this library to work with the server. It only needs to send messages in a form recognized by the server (HTTP 1.1/SLTP).

```
// sltpclient.C example of a simple simlink client
//
// Kevin Brown Dec. 1999
//
// to run, assuming the server is located on
// nserver.xxx.yyy.zzz and is using port
// number 9669:
//
// % sltpclient 9663 nserver.xxx.yyy.zzz
#include <pazu.H>
#define BLKSIZE 512
char* MSG1=
"GET / SLTP/1.0\nConnection:
Keep-Alive\nHost: nserver.xxx.yyy.zzz\n\n";

int main(int argc, char* argv[])
{
    int port=-1;
    int outfd;
    ssize_t bytesread;
    ssize_t byteswritten;
    char buf[BLKSIZE];

    memset(buf,0,sizeof(buf));
    if (argc < 2) {
        fprintf(stderr,
            "Usage: %s <port> [host] \n", argv[0]);
        exit(1);
```

```

}
(void)sscanf(argv[1], "%d", &port);
char *host=(port==-1)?argv[1]:argv[2], socknm[80];
P_IO* s_io=new P_IO(
    "socket",
    port!=-1?AF_INET:AF_UNIX,SOCK_STREAM);

if((outfd=s_io->p_connect(host,port))<0){
    s_io->p_error(
        "Unable to establish an Internet Connection.");
    exit(1);
}
fprintf(stderr,
"Client: Connection has been made to %s\n",
    argv[2]);

// write uses the file descriptor contained in the
// s_io class. This is what the -1 indicates.
s_io->p_write(-1, MSG1, (int)strlen(MSG1)+1);

//wait for message to come back
while((bytesread=
    s_io->p_read(outfd, buf, sizeof(buf)))>1)
{
    cout << buf ;
    memset(buf,0,sizeof(buf));
}
cout << endl;
s_io->p_close(outfd);
}

```

VII. SUMMARY

In the present implementation the system works very well. The testing of the system has included testing all the command line arguments (using a different socket port, using a different configuration file) and testing connections to simulators via sockets and fifos. For the bnlmad simulator [1,2], which takes a few seconds to a few minutes to run (depending on the mode and model) the added latency of going through the server is very small. The data integrity is excellent. For the same run, using the bnlmad simulator, the output data taken through the server was identical to the data written by the simulator when run by itself. We have not seen any data corruption resulting from being filtered through the server.

Here a normal bnlmad run takes a few seconds.

```

265% time bnlmad -da agsmad.config
Normal End of Program.
1.683u 0.225s 0:03.21 59.1% 0+0k 94+1io 53pf+0w

```

The same run, through the server takes a few seconds longer. This overhead is constant. When doing longer simulation runs, the overhead remains a few seconds.

```

134% time bin/sltclient 9669 nserver > runout
Client: Connection has been made to nserver
0.017u 0.054s 0:06.64 0.9% 0+0k 0+1io 0pf+0w

```

For the BNL model server project this is a very important development. The ability to connect other models to the control system will enable us to start working on automating various processes in the accelerator that currently cannot be automated. Additionally we can now provide more precise model data to applications using simplistic internal models.

ACKNOWLEDGMENTS

This work was supported by the U.S. Department of Energy. We wish to thank Todd Satogata (BNL C-AD) for offering valuable advice and stimulating discussions.

-
- [1] K.Brown et al, "The RHIC/AGS Online Model Environments: Experiences and Design for AGS Modeling", 1999 US Particle Accelerator Conference Proceedings. No. 2722
<http://pac99.bnl.gov/>
 - [2] T.Satogata et al, "The RHIC/AGS Online Model Environments: Design and Overview.", 1999 US Particle Accelerator Conference Proceedings. No. 2728
<http://pac99.bnl.gov/>
 - [3] See the Automated Beam Steering and Shaping home page at CERN:
<http://www.cern.ch/ABS/>
 - [4] See the main cdev web page at Jefferson Lab.:
<http://www.jlab.org/cdev/>
 - [5] The Apache Software Foundation main web page is found at:
<http://www.apache.org/>
 - [6] "Practical Unix Programming: A Guide to Concurrency, Communication, and Multithreading", K.Robbins and S.Robbins, Prentice Hall PTR, 1996.
ISBN 0-13-443706-3
 - [7] "UNIX System Programming Using C++", T. Chan, Prentice Hall PTR, 1997.
ISBN 0-13-331562-2
 - [8] See RFC 2616 for a complete description of http/1.1 protocol.
<http://www.w3.org>
 - [9] "unix Network Programming, Networking API: Sockets and XTI", Volume 1 2nd Ed., W.Richard Stevens, Prentice Hall PTR, 1998.
ISBN 0-13-490012-X
 - [10] "unix Network Programming, Interprocess Communications", Volume 2 2nd Ed., W.Richard Stevens, Prentice Hall PTR, 1999.
ISBN 0-13-081081-9