CDMS Note 9712002 2^{nd} edition

The CDMS Net Protocol & GPIB Protocol User's Guide

Steve Eichblatt, Fermilab
Sae Woo Nam, Stanford University

1 Introduction

The original motivation for this program was to write an intelligent LabView script reader to send commands to the GPIB box. The possibility to use a scripting language such as perl in conjunction with LabView seemed attractive. In the process we found the TCP/IP functions of LabView, which enable users to pass strings to a LabView program from anywhere on the internet.

Communication between processes running on different machines enables the group of machines to be treated as a cluster. This allows multiple users simultaneously on the same machine, even if that machine is a Macintosh. Interprocess communication also allows files to be stored in a single location and accessed by everyone, and potentially allows users to split large jobs into pieces and distribute the jobs across the cluster of machines.

The "`GPIB Protocol`" is the language or format used to issue commands and read back responses to and from a GPIB Box connected to a machine running the `GPIBserver`.

The "`Net Protocol`" is the language, or precise format that is used to extract individual events, entire files, or directory information from a machine running the `netServer`. It can be used as a remote file server, similar to nfs, and also a web page server, similar to httpd.

The protocols are very new, and are likely to increase in functionality. This note is intended to describe what the servers are capable of, and to be used as a how-to document for writing clients which use the servers through these protocols. Readers are invited to write clients and make suggestions. The source code for the programs mentioned in this article can be found at
`http://fnphyx-www.fnal.gov/experiments/cdms/software/net`.

2 Design

The design of the `netProtocol` and the `GPIBProtocol` are based on the client/server model. In this model, a *server* runs on one machine listening on a given *port* for *clients* to open a *socket*, or connection, and to give the server instructions. Figure 1 shows An example of possible interconnections between the computers in the CDMS cluster.

2.1 The Servers

The server takes no action unless instructed by a client. The server can serve clients coming from any location connected to the server's host machine (including from the server host itself). For security the server may refuse service to any client that does not meet any standard the server cares to impose, such as providing a valid password. The servers perform a limited set of operations, described below.

The netServer

- serve files.
- inflate and serve `gzip`ped files.
- serve single events from large event files.
- inflate and serve single events from `gzip`ped event files.

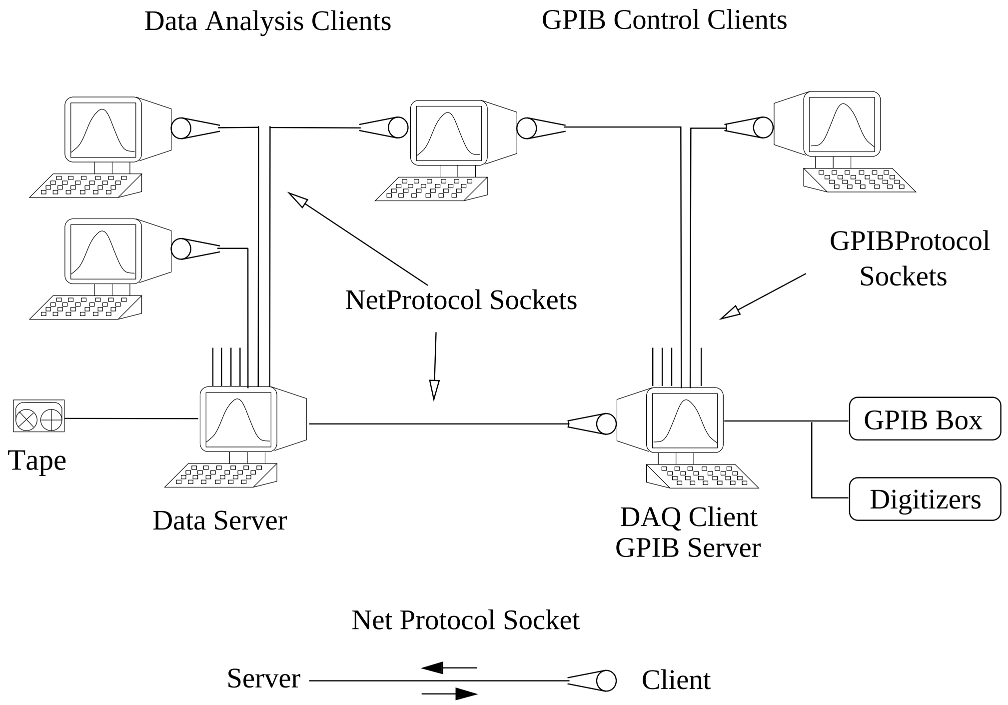

Figure 1: *A possible configuration of machines connected by* `netProtocol` *and* `GPIBprotocol`*. The sockets are shown with the Client on the wide end of the line. The server has the potential to run listen for many sockets simultaneously.*

- give a directory listing (ls).
- receive files (with pathname) and writes them to disk.
- receive files (with pathname), compress and write them to disk. (not yet).
- receive events, compress and/or append them to an event file.

The GPIBServer

- write to GPIB and return status.
- read from GPIB and return data and status.

The `netServer` is multi-threaded so it can serve multiple clients simultaneously by spawning a separate process to deal with client's requests while continuously listening for incoming requests.

2.2 The Protocols

The `protocol` is the agreed-upon *lingua franca* between clients and servers. The protocol is defined by the server and specifies what the server will and will not do.

Protocols are built on other protocols. The basic communication mechanism between computers can be TCP/IP (internet protocol), UDP, etc. Both `netProtocol` and `GPIBprotocol` are based on TCP/IP for maximum reliability and universal compatibility.

2.2.1 `netProtocol`

The netProtocol server is launched by typing `java netServer` (Be sure that the netServer.class file is in your `$CLASSPATH`. The command line options are:

-h ¡hostname¿ Only allow clients from ¡hostname¿ to connect to the server.

-p ¡port¿ Run the server on port ¡port¿. Default is 4444.

-v Prints the version number to the screen and exits.

-w Allow clients to send data to be written to files on the server machine. (default off).

The list below summarizes the commands understood by `netServer`. All commands are terminated with a newline (\n) before the first argument is passed.

All States

BYE
Closes connection Output: `"BYE\n"`
Final State: Connection Closed.

SETSTATE String `newState`
Put the server in state `newState`
Output: `"OK"`
Final State: `newState`.

BadCommand
Output: `"Bad command"`
Final State: *unchanged*.

State = Waiting

FILENAME String `filename`.
Specify a filename for later use in the `GET` and `DATA` functions.
Output: `"GOTFILENAME\n","NO SUCH FILE\n"`.
Final State: GotFileName.

MKDIR String `dirname`.
Make a directory named `dirname` on the server.
Output: `"ACK\n"`.
Errors: `"BAD MKDIR\n"`.
Final State: Waiting.

PWD
Returns the server's current working directory.
Output: String `pathname`
Final State: Waiting.

CD String `dirname`.
Changes the current working directory on the server.
Output: `"ACK\n"`
Errors: `"NO SUCH DIRECTORY\n"`
Final State: Waiting.

LS String `dirname`.
Returns a listing of the files in the directory `dirname`.
Output: int `nFiles`,String `filename[nFiles]`
`"ACK\n"`
Errors: `"NO SUCH DIRECTORY\n"`
Final State: Waiting.

GET String `HTTPRequest`.
Web-browser compatible file request. This option makes files on the server visible to web browsers which specify the URL of the file, and the port number of the server.
Output: The file in the format expected by the browser. See the source code or documents on HTTP for details.
Final State: Waiting.

State = GotFileName

GET
Gets the file specified by previous `FILENAME` command.
Output: int `filesize`, byte `data[filesize]`,`"OK!\n"`
Errors: `"INVALID LENGTH FILE\n"`,`"NO SUCH FILE\n"`,`"NO PERMISSION\n"`,
`"FILE ERROR\n"`,`"IO ERROR\n"`
Final State: Waiting.

ZGET
Uncompresses and gets the `gzip`ped file specified by the previous `FILENAME` command.

Output: int `packetSize`, byte `packet[packetSize]`,
int `packetSize`, byte `packet[packetSize]`,
int `packetSize`, byte `packet[packetSize]`,
...
int -1 (indicates EOF)
`"OK!\n"`

Errors: `"NO SUCH FILE\n"`,`"NO PERMISSION\n"`,`"IO ERROR\n"`
Final State: Waiting.

FOPEN
Open the file specified in `FILENAME` for reading.
Output: "OPEN"
Errors: "ERROR IN FOPEN"
Final State: FileOpen.

FOPEN(W)
Open the file specified in `FILENAME` for writing.
Output: "OPEN"
Errors: "No permission. Ha."
Final State: FileWrite.

GETEVENT Int `begin`, Int `nevents`.
Extract events `begin` to `begin+nevents` from the file specified in `FILENAME`. If the file is `gzip`ped, then the server will uncompress the data before sending it.

Output: Int `FirstEventSize` Byte `data[FirstEventSize],`
Int `SecondEventSize` Byte `data[SecondEventSize],`
...
Int `Nevent`th`Eventsize` Byte `data[`Neventth`EventSize],`
Int `-1` *indicates end of ev*`"OK!\n"`

Errors: `"NO SUCH FILE\n","NO PERMISSION\n","ERROR",`
`"ERROR READING EVENT\n"`
Final State: Waiting.

GETBYTES Int `origin`, Int `length`.
Get `length` bytes starting at `origin` from file specified by previous `FILENAME` command.
Output: Int `nbytes`, Byte `data[nbytes], "OK\n"`
Errors: `"NO SUCH FILE\n","NO PERMISSION\n","ATTEMPT TO READ PAST EOF\n",`
`"ERROR\n","FILE ERROR\n","IO ERROR\n"`

PUT Int `filesize`, Byte`data[filesize].`
Puts the data into a file in the current working directory on the server. The file name is specified by the previous `FILENAME` command.
Output: `"ACK\n"`
Final State: Waiting.

DATA Int `filesize`, Byte `data[filesize].`
Identical to the `PUT` command. Maintained for backwards compatibility.

APPEND Int `size`, Byte `data[size].`
Appends the byte array `data` to the end of a file in the current working directory on the server. The file name is specified by the previous `FILENAME` command.
Output: `"ACK\n"`
Final State: Waiting.

State = FileOpen

FREAD Int `skip`, Int length
Read the next `length` bytes starting `skip` bytes from the last byte read from currently open file.
Output: Int `nbytes`, Byte `data[nbytes], "OK\n"`
Errors: `"FILE ERROR","IO ERROR"`
Final State: FileOpen.

FCLOSE
Close the currently open file.
Output: Int `"CLOSED"`
Final State: Waiting.

GETEVENT Int `skip`, Int `nevents`. *not yet impemented!!*
Extract `nevents` events starting at `skip` events from the end of the last event read from the open file. If the file is `gzip`ped, then the server will uncompress the data before sending it.

Output: Int `FirstEventSize` Byte `data[FirstEventSize],`
Int `SecondEventSize` Byte `data[SecondEventSize],`
...
Int `Nevent`th`Eventsize` Byte `data[`Neventth`EventSize],`
Int `-1` *indicates end of ev*`"OK!\n"`

Errors: `"NO SUCH FILE\n","NO PERMISSION\n","ERROR",`
`"ERROR READING EVENT\n"`
Final State: FileOpen.

State = FileWrite

FWRITE Int length Byte `data[length]`
Write `length` bytes to the currently open (for writing) file.
Output: `"ACK"`
Errors: `"ERR"`
Final State: FileWrite.

FCLOSE
Close the currently open file.
Output: Int `"CLOSED"`
Final State: Waiting.

2.2.2 `GPIBProtocol`

gpibread Int `bytesToFollow`, Int `GPIBAddress`, Int `Mode`, Int `timeout`, Int `nbytes`.
Reads `Data` from the GPIB address `GPIBAddress`.
The variable `bytesToFollow` is ignored, as it is always 16. It is included for compatibility witht he *gpibwrite* command.
It is the client's responsibility to set the `Mode` corresponding to the device being read, and to prepare the device to send out data.
Output: Int `size`, Byte `Data[size]`, Int `StatusLength`, Byte `Status[StatusLength]`,
NOTE: Current LabView implementations of the server return a different output (which should be fixed). They return Int `size` Byte `Status+Data`, where `Status+Data` are the status and data words. The status word is 4 bytes long, followed by the Data array which occupies the remainder of `size`.

gbipwrite Int `bytesToFollow`, Int `GPIBAddress`, Int `Mode`, Int `timeout`, Int `nbytes`
Byte`Data[bytesToFollow-16]`.
Sends `Data` to the GPIB address `GPIBAddress`. Clients are responsible for sending valid data to the GPIB device of interest.
`nbytes` is ignored by the server. It is included for compatibility with the *gpibread* command.
Output: Int `StatusLength`, Byte `Status[StatusLength]`.

bye
Closes the connection to server.

2.3 Sockets

Communication between servers and clients takes place over a `socket`. Reading from and writing to a socket is analogous to reading to and writing from a file. In fact, most computer languages use exactly the same syntax. A socket is treated like a file in high-level programming languages. It is important to note that opening, closing, reading and writing to sockets (and files) depends on the language you are writing in.

2.4 Clients

The role of the client is clearer if we rename the "Client/Server" model to the "Master/Slave" model. The client tells the server what to do. The client tells the server in the language of the protocol.

An important feature of the client/server model is that the client can be written in any language, as long as it passes commands to the server in the correct protocol. Both protocols described here are built upon TCP/IP, so nearly all computers can send commands to the servers.

Section 4 contains extracts of source code for several clients. For complete documentation, see the website `http://fnphyx-www.fnal.gov/experiments/cdms/software/net`.

3 Performance

One of the initial pleasant surprises encountered with the `netProtocol` program was the data transfer speed across the network. Figure 2 shows the data transfer speed as a function of package size. The slow speeds for small packages is due to the connection establishment and protocol commands. This data was using a labView client on `qet.stanford.edu` to send data to the server on `hep.stanford.edu`. Transfer speeds using the `ftp` program are also shown, although the comparison is unfair, since `ftp` involves an additional disk-read on the client machine, which could dominate the transfer speed (or perhaps `ftp` limits the packet size to 10 kbytes).

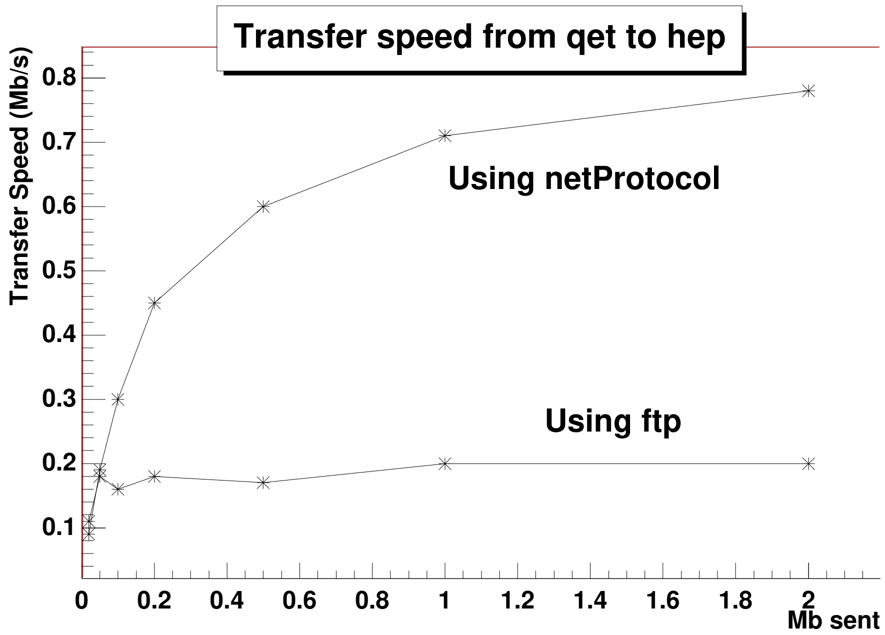

Figure 2: *Transfer rate as a function of packet size. The comparison is not fair, since* `netProtocol` *transfers are directly from memory, while the ftp transfer is from disk to disk. The comparison does illustrate the penalty paid by using* `ftp`.

4 Code

Actual code may illuminate this description. We have tried to choose code which is clear at the cost of not including everything. For complete program listings, see the website
`http://fnphyx-www.fnal.gov/experiments/cdms/software/net`.

4.1 processInput

The definition of the netProtocol is in the code below. The routine `processInput` is part of the server. In java, the code can be shared and used by the clients, which redefine the methods.

```
// ------------------------------------------------------- processInput
  String processInput(String In, DataInputStream instream,
                      DataOutputStream outstream) {
    theInput = In; is = instream; os = outstream;
    String theOutput = "BAD";
    String inputLine = null;

    if (theInput != null) System.out.println ("Received "+theInput);

    Date foo; foo = new Date();

    switch (state) {
    case HELLO:
      state = WAITING;
      if (theInput != null) { return ("Hello"); }
      else { return("");}
    case WAITING:
      if         (theInput.equalsIgnoreCase("FILENAME")) {
        return (readFilename());
      } else if (theInput.startsWith       ("GET")    ) {       // http GET
        return (replyHTTP());
      } else if (theInput.equalsIgnoreCase("MKDIR")   ) {
        return (this.mkdir());
      } else if (theInput.equalsIgnoreCase("CD")      ) {
        return (this.cd());
      } else if (theInput.equalsIgnoreCase("PWD")     ) {
        return (this.pwd());
      } else if (theInput.equalsIgnoreCase("LS")      ) {
        return (ls());
      } else if (theInput.equalsIgnoreCase("GPIB")    ) {
        return (gpib());
      } else if (theInput.startsWith("gpib")          ) {
        return (gpib());
      } break;
    case FILEOPEN:
      if         (theInput.equalsIgnoreCase("FCLOSE")   ) {      // close the file
        return (fclose());
      } else if (theInput.equalsIgnoreCase("FREAD")    ) {
        return (fread());
      } else if (theInput.equalsIgnoreCase("FWRITE")  ) {      // WRITE to open file.
        return (fwrite());
      } else if (theInput.equalsIgnoreCase("GETEVENT")) {      // GET Event from data file
        return (getEvent());
      } break;
    case FILEWRITE:
      if         (theInput.equalsIgnoreCase("FCLOSE")   ) {      // close the file
        return (fclose());
      } else if (theInput.equalsIgnoreCase("FWRITE")  ) {      // WRITE to open file.
        return (fwrite());
      } break;
    case GOTFILENAME:
```

```
      if        (theInput.equalsIgnoreCase("DATA")    ) {      // PUT file
        return (saveFile());
      } else if (theInput.equalsIgnoreCase("PUT")     ) {      // PUT file
        return (saveFile());
      } else if (theInput.equalsIgnoreCase("APPEND")  ) {      // APPEND to file
        return (saveFile(true));
      } else if (theInput.equalsIgnoreCase("GET")     ) {      // GET file
        return (getFile());
      } else if (theInput.equalsIgnoreCase("ZGET")    ) {      // GET zip file
        return (getZipFile());
      } else if (theInput.equalsIgnoreCase("GETEVENT")) {      // GET Event from data file
        return (getEvent());
      } else if (theInput.equalsIgnoreCase("GETBYTES")) {      // GET a set of bytes
        return (getBytes());
      } else if (theInput.equalsIgnoreCase("FOPEN"))    {      // open file
        return (fopen());
      } else if (theInput.equalsIgnoreCase("FOPEN(W)"))    {      // open file
        return (fopen(true));
      } break;
    }
    if        (theInput.equalsIgnoreCase("BYE")     ) {
       return (bye());
    } else if (theInput.equalsIgnoreCase("SETSTATE")) {
       return (setState());
    } else {
      return (badCommand());
    }
}
```

4.2 getEvent

Below is a client's method `getEvent`. It calls the server's `GETEVENT` function. It can only be called if the `FILENAME` has beed defined. The routines `sendInt`,`recvInt`,`recvFully`,`recvText` perform the direct communication with the servere. These routines are defined elsewhere in the program and not shown here (see website).

```
// -------------------------------------------------------- getEvent
protected String getEvent(){    // a general-purpose Client to get an Event.
  state=WAITING;
  sendText("GETEVENT");

  int beginLength[] = new int[2];   //  beginning event number and number of events
  beginLength[0] = Integer.parseInt(readInput());     // read from kbd.
  beginLength[1] = Integer.parseInt(readInput());     // read from kbd.
  sendInt(beginLength);              // send first event and number to server.

  int size = recvInt();             // read size of buffer from server.
  System.out.println("begin, length, size "+beginLength[0]
                   +" "+beginLength[1]+" "+size);
  while (size>0){
    byte b[]= new byte[size];
```

```
    recvFully(b);                   // read buffer from server.
    try {
      System.out.write(b,0,size);   // write buffer to screen.
    } catch (Exception e){
      System.err.println("Exception "+e);
    }
    size = recvInt();
  }
  return(recvText());               // pass back the return string from the server.
}
```

4.3 MATLAB client

Here is a more detailed example. It is a MATLAB client that reads an event from a file.

```
/*
** Matlab client demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include "mex.h"
#include "netMexLib.h"
#define PORT 4444    /* the port client will be connecting to */

#define MAXDATASIZE 100 /* max number of bytes we can get at once */
#define SOURCENAME  prhs[0]
#define START  prhs[1]
#define NUM_EVENTS  prhs[2]

void mexFunction(
        int             nlhs,
        mxArray *plhs[],
        int             nrhs,
        const mxArray   *prhs[]
        )

{
    int sockfd, numbytes, bytes_sent, bytes_recv,counter;
    char **buf, msg[100], *temp, hostname[100];
    char *sname, filename[255], command[1000];
    int port,n, start, num_events;
    int events_recv, event_counter, *len;
    struct hostent *he;
    struct sockaddr_in their_addr; /* connector's address information */
```

```
    FILE *f;

    if (nrhs != 3) {
        mexErrMsgTxt("Wrong number of parameters\n");
    }

    n = mxGetN(SOURCENAME) + 1;
    sname = mxCalloc(n,sizeof(char));
    mxGetString(SOURCENAME,sname,n);

    start = (int) mxGetScalar(START);
    num_events = (int) mxGetScalar(NUM_EVENTS);

    printf ("start: %d\t num_events: %d\n",start,num_events);
    if (parse_ntp(sname, hostname, &port, filename)!=0)
      mexErrMsgTxt("Bad ntp specification");
    printf("%s\t%d\t%s\n",hostname,port,filename);
    if ((he=gethostbyname(hostname))== NULL) {  /* get the host info */
        perror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        mexErrMsgTxt("socket");
    }

    their_addr.sin_family = AF_INET;         /* host byte order */
    their_addr.sin_port = htons(port);     /* short, network byte order */
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    bzero(&(their_addr.sin_zero), 8);        /* zero the rest of the struct */

    if (connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct
sockaddr)) == -1) {
        mexErrMsgTxt("connect");
    }
    sprintf(command,"filename\n%s\n",filename);
    if ( (bytes_sent = send_msg(sockfd, command)) < 0) {
      mexErrMsgTxt("send");
    }

    if ( (bytes_recv = recv_msgln(sockfd, msg, MAXDATASIZE)) < 0) {
      mexErrMsgTxt("recv");
    }
    printf ("recv %d:%s\n",strlen(msg),msg);
    if (strcmp(msg,"GOTFILENAME\n")!=0) {
      bytes_sent = send_msg(sockfd,"bye\n");
      mexErrMsgTxt("No such file exists");
    }
    if ( (bytes_sent = send_msg(sockfd, "GETEVENT\n")) < 0) {
      mexErrMsgTxt("send");
```

```
    }
    if ( (bytes_sent = send_bytes(sockfd,&start,sizeof(int))) !=4) {
      mexErrMsgTxt("send:start");
    }
    if ( (bytes_sent = send_bytes(sockfd,&num_events,sizeof(int))) !=4) {
      mexErrMsgTxt("send:num_events");
    }

    buf = mxCalloc(sizeof(char *),num_events);
    if ( (events_recv = recv_events(sockfd, buf, num_events)) < 0) {
      mexErrMsgTxt("recv");
    }
    printf ("events_recv: %d\n",events_recv);
    temp = *buf;
    f = fopen("/tmp/temp.mat","wb");

    for (event_counter=0;event_counter<events_recv;event_counter++) {
      temp = buf[event_counter];
      len = (int *)temp;
      for(counter=0;counter<*len;counter++) {
        fputc(*temp++,f);
      }
    }
    fclose(f);

    close(sockfd);

}
```

4.4 GPIBtest.pl

Finally, a client which uses the `GPIBserver`. This client is a perl class. Again only relevant pieces are shown. The low-level class definition is available on the web for the interested reader.

```
#! /usr/local/bin/perl -d

use GPIBinstr;               # instruction class
use GPIBconnection;          # connection class

# create a GPIBconnection to host, port.
$server = new GPIBconnection("hostname.fnal.gov:1234");

$inst = new GPIBinstr("gpibwrite",18,"0000");   # create a GPIBinstruction

for ($i=0;$i<10;$i++){            # useless loop for demonstration.
  $data = 0xfa + $i;
  $inst->setData(sprintf "%04x",$data);
  $inst->print;                   # print the instruction to the screen.
  $server->exInst($inst);         # execute the instruction on server.
}
```