CDMS note 9807001 July 22, 1998

A Sketch and Simple Example of a Net Based Object-Oriented Slow Control System for CDMS

Steve Eichblatt, Fermilab

1 Introduction

Those who have spent a day in the Soudan mine tend to favor a remote controlled system for writing to and reading from the electronics, while those concerned about security favor the net-blind solution. I feel that we need a net friendly DAQ and slow control. This paper describes the design and a partial object-oriented implementation of a net-friendly slow control system. The design is intended as something to consider for future runs or CDMS II. The particular perl-based implementation described here is one of a class of solutions and I do not reccommend it above others.

2 Idea

The idea described here is to pass GPIB commands to the GPIB box *via* the internet ultimately using a client such as Netscape or telnet to issue commands. Figure 1 shows the elements and interconnections.

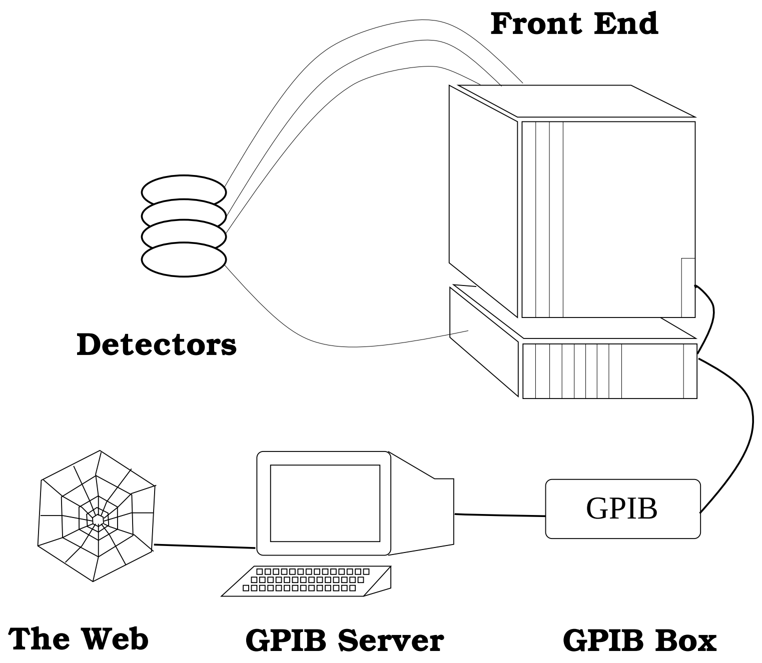

Figure 1: *Configuration of the Slow (GPIB) Control of the CDMS electronics at Fermilab.*

In this scheme, the user tells the client (*e.g.*Netscape) what command he wants to perform on the front end electronics. The client forms a request and sends it *via* the web to the server, which then issues a GPIB command to the GPIB interface box, which sends the command to the crate controller and onto the cards. A read operation will return data from the crate to the GPIB server and back to the client who originated the request.

Compared with a non-network solution there are a few additional layers of intercommunication to perform any action.

3 Framework

3.1 The Objects (and a bit of jargon)

An object-oriented approach is natural for controlling hardware where object definitions align with physical objects. Figure ?? is a diagram of the objects used in implementing the design, and their interconnections. Each name within a cloud represents an *object*, which can be thought of as a variable type containing both data and methods (or subroutines). Each object is usually defined in a separate file.

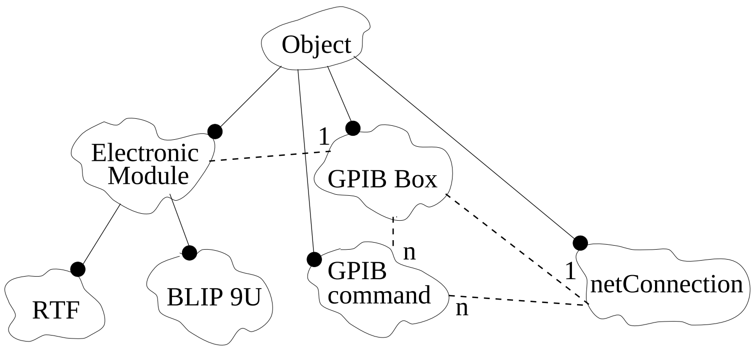

Figure 2: *A diagram of objects in the slow control system. Solid lines represent an 'is-a' relationship, and the dotted lines are 'has-a' relationships.*

A dotted line in figure 2 indicates a *'has a' relationship*, also known as an *aggregate*. The number at one end of the line tells the number of objects contained. Specifically it means that the class `Electronic Module` has, as part of its data, one variable of type `GPIB box`, which in turn has a `netConnection` as a data member.

A solid line indicates an *'is-a' relationship*, or *inheritance*. The solid black dot marks the sub-class. Any data members or methods defined in a parent class are available in the inherited classes. All of the objects shown inherit from the `Object` class, directly or indirectly. So anything that an `Object` can do can be done by any of the classes. For example the Print command is defined in `Object` and prints the name of the class.

The `GPIB box` class is the analog of Dennis's GPIB Interface box. the `Electronic Module` passes it arguments such as `gpibwrite 0040 0FFF`, and it creates a `GPIB command` which is a pure GPIB string. The data members of the `GPIB box` are things such as the GPIB address of the box (18), and the GPIB headers and tails.

The `netConnection` class 'talks' to the GPIB server. Its input is a `GPIB Command`, Its methods translate it to a set of strings and lengths, sends them to the server and receives the status in the language of the GPIB protocol[1]. Its data members include the hostname and port number of the server.

The similarity of figures 2 and 1 are obvious. The advantage of an object oriented programming language is that the same similarity exists between the object diagram and the code that implements it. This makes an easy mapping from code to concept and vice-versa, so the code is more readable and therefore more flexible.

3.2 Getting it on the Web

Once you have a standalone executable program, it is a simple matter to create a web form that runs the program using the *Common Gateway Interface*, or cgi programming [3]. The idea is to write a program which reads its inputs from a web request, and returns a web page for the browser to display.

So the user interface is a web page, which everyone knows and loves. Figure 3 shows the web page for doing the low-level read and write to the GPIB box, *à la* Dennis's vi.

Figure 3: *The web page that does "Dennis' vi".*

When the user hits the "send" button, the server calls a perl program like the one below with the dots filled in.

```
 use CGI ;
 use GPIBbox ;                                    # link in my class definitions.

# read the arguments and fill the variables:
#   host, port, subrack, address, data, and readWrite

    ...   ...   ...

# create a GPIB box object

    $box   = new GPIBbox("$host:$port");
    $box->subrack($subrack);

# read or write the data/address to the GPIB box.

    if ($query->param("readWrite") =~ "read") {
      $box->readData($address);
    } else {
      $box->writeAddress($address);
      $box->writeData($data);
```

```
    }

# send back the resulting (html) page

      ...   ...   ...
```

This simple low-level program demonstrates that we can do all of the slow control over the web, if we choose.

4 Implementation

4.1 Existing prototype

I have written an working example of this technique using a LabView GPIB server (see [1] for details) and a client in perl ([2]).

This is not the appropriate place to discuss the details of the perl implementation. Suffice it to say that there is one that works for the BLIP, RTF, and ZIP 9U boards. I have not yet written a document describing the netConnection and the GPIB box classes, but there are write-ups on the BLIP, RTF, and ZIP objects. Refer to [4] for a description of the perl based solution for these boards, and where to download all of the code.

4.2 Running Remotely = Efficiency

Since the program runs on a unix machine, it can be run from any location *via* telnet. It can also be edited from any location *via* telnet. While this may seem like a trivial advantage, it can make a huge difference it efficiency. It allows debugging of the hardware and software to happen in parallel.

4.3 Why perl?

I chose to write the client in perl since it is an interpreted language. I was happy to pay the penalty in execution speed for the lack of compiling.

As well as being a scripting language, Perl has a simple interactive interpreter. This allows the user to interact with the electronics one command at a time. When a series of commands are found to be useful, they can be extracted from a history file and turned into a script. The user can take advantage of all the features of perl, as well as the object described above.

If we decide to write the DAQ in another object oriented language, such as C++, then it would make sense to write the Slow control in the same language. This would allow us to pass objects directly from the slow control system to the DAQ. In this way, the data stream could easily contain the electronics objects, and be inquired to know the gain and ID of each module. This can be done from perl, but it would be more transparent in a common language.

Writing the program in Java would make the Netscape interface really slick, with animated buttons, and movie frames cycline while the transaction is taking place. It would also run faster than the perl version.

For the first prototypical example, I chose perl for its simplicity and for the rapid-development that comes with an interpreted language. If we decide to adopt an object oriented slow control system, we are *not* bound to use perl.

5 Summary

I have presented a framework and a working prototype of an object-oriented net-friendly slow control system.

This solution increases efficiency by allowing hardware and software debugging to proceed in parallel. And once debugged, it allows the remote control of an experiment which is to be built in an inaccessible location. An object-oriented approach is natural for the task, and it eases the maintenance of the code thanks to the similarity between the design and the implementation.

When the time comes to decide how to do the slow control in future, let's keep these ideas in mind.

References

[1] Steve Eichblatt, Sae Woo Nam CDMS note 9712002. *The CDMS Net Protocol*

[2] see *e.g. Programming Perl* by Larry Wall and Randal Schwartz, O'Reilly & Associates, Inc.
or simply `man perl` on a unix system.

[3] Shishir Gundavaram, *CGI programming on the World Wide Web* O'Reilly & Associates, Inc., 1996

[4] Steve Eichblatt & Steve Furlanetto,
The BLIP & RTF Card Drivers, CDMS Note 9807002
`http://fnphyx-www.fnal.gov/experiments/cdms/software/GPIB/Drivers.ps`.
The ZIP Card Driver
`http://fnphyx-www.fnal.gov/experiments/cdms/software/GPIB/ZIPdriver.ps`.