Glish: A Software Bus for High-Level Control

Vern Paxson
Lawrence Berkeley Laboratory
One Cyclotron Road
Berkeley, CA 94720 USA

Abstract

Glish is a software system for building high-level control applications out of modular, event-oriented programs. Glish provides these applications with a high degree of flexibility, so they can adapt quickly to changing requirements. We describe the strengths of the "software bus" approach, how Glish can direct and modify interprocess communication within a distributed application, and how the system is currently used for orbit-correction at the Advanced Light Source at LBL.

1 Introduction

We call an accelerator control application "high-level" if it deals with the accelerator in terms of the underlying physics rather than the underlying hardware. One key aspect of such applications is that in general they do *not* require rapid ($\leq$ 1 msec.) real-time response times. Some examples are correcting the orbit, controlling the chromaticity, and changing the tune. These applications have a number of characteristics that can make writing them difficult:

1. The applications tend to be complex, requiring connecting together disparate elements (user interface, physics algorithm, data acquisition, data archiving, hardware control;
2. The different components of an application might need to run on different computers and communicate over a network;
3. Often the applications must incorporate existing software written by different people using differing conventions;
4. An application might need to run in different "modes", such as on-line, off-line exploration of archived data, and simulation;
5. An application might need to work with different accelerators (e.g., booster vs. storage ring);
6. The applications often play key roles in day-to-day accelerator operations, making it vital that they remain resilient in the face of failures, and can be rapidly modified to accommodate changes in operational procedures.

The *Glish* software system attempts to address each of these issues. The fundamental idea behind Glish is to build such applications by connecting together a set of modular, event-oriented Unix programs. These programs are referred to as *clients*. Glish provides a "software bus" for directing and modifying the data exchanged between programs, as well as facilities for creating and controlling processes across a network of possibly heterogeneous computers. A key aspect of Glish is that control of the "software bus" is done not via C or C++ code, but using a high-level scripting language that promotes rapid creation and modification of applications.

In the next section we discuss what we mean by "event-oriented" programs and how using them ensures a high degree of modularity. The next two sections present an overview of the Glish scripting language used to connect together Glish clients, and discuss the "under-the-hood" details of interprocess communication, including performance measurements. We then look at how clients use the Glish C++ Client library to connect to the Glish system, and at how to integrate existing non-Glish programs into the system. We next discuss how Glish deals with client failures and network outages, and finish with an example of a large Glish application.

2 "Event-Oriented" Programs

A central concept in Glish is that of an *event*. An event is an arbitrary, typed data value and an associated name. For example, an event might be an array of floating-point numbers along with the name `BPM_readings`.

Every client used in a Glish application is written in an "event-oriented" style. That is, clients are written in terms of what events they expect to receive, and what events they generate. For example, an FFT client might expect to receive a `do_fft` event whose value is an array of floating-point values, and it in turn generates an `answer` event whose value is a record containing two fields, `real` and `imag`, each an array

giving the real and imaginary parts of the Fourier transform of the original data.

A crucial point regarding event-oriented programs is that they do *not* have any knowledge of where their events come from, or where their events go to. As far as each individual Glish client is concerned, its incoming events arrive from some black box, and it ships its outgoing events to another black box. Glish does *not* provide any mechanism for clients to know what other clients are being used in an application. Thus, Glish's event-oriented style assures the complete modularity of every client used in the system.

The natural question then arising is, How do you connect together clients to form an application when you can't code into them knowledge of other clients? The answer is that such connections are made using the Glish scripting language.

3 The Glish Scripting Language

The particulars of which clients are used to build a distributed Glish application, what hosts they run on, and how they communicate their events with one another, are all controlled by scripts written in the Glish language. This language is described in greater detail in [1]; here we endeavor to convey its general feel and power.

The Glish language is quite high-level and makes it easy to create and manipulate client processes. For example, the Glish statement:

```
bpms := client("read_BPMs")
```

creates a Glish client by running the Unix program `read_BPMs` on the local host. It then assigns to the variable `bpms` a value corresponding to this client; the variable can be subsequently used to refer to the client and to send events to it and receive events from it. To create the client on the remote host `vme21`, we could instead use:

```
bpms := client("read_BPMs", host="vme21")
```

The `client()` function is simply a predefined Glish function; the use of `host=` to specify the remote host is just an instance of the general Glish "named argument" feature, available to any function, whether built-in or written by the user.

Once we have created a client, we can send it an event using the `send` statement:

```
send bpms->do_read(crate=5, plane="X",
                   calib=[0.01, -0.5])
```

sends a `do_read` event to the `bpms` client. The value of the event is a record with three fields, `crate` (equal to the integer `5`), `plane` (equal to the string `X`), and `calib` (equal to an array of two floating-point values).

Suppose that when it receives a `do_read` event, `read_BPMs` reads the BPM values from the given crate, applies the given calibrations, and then generates a `read_done` event whose value is a floating-point array of the calibrated BPM readings. We could print out the results in our Glish script using:

```
whenever bpms->read_done do
    print $value
```

When executed, the `whenever` statement indicates that any time the `bpms` client generates a `read_done` event, execute the statement `print $value`. `$value` is a special Glish expression meaning, "the value of the last event received".

In general, a `whenever` statement can specify an *arbitrary* action in response to any event: the statement executed can include function calls, sending out new events, creating new clients, loops, or activating more `whenever` statements. One common action for a `whenever` statement is to send the received event to another client. For example, suppose we have a plotting program that we created using:

```
plot := client("plotter",
               "-xaxis BPM number",
               "-yaxis BPM reading")
```

and that this client updates its display whenever it is sent a `new_plot` event. Then we could maintain a display of the current BPM readings using:

```
whenever bpms->read_done do
    send plot->new_plot($value)
```

Now suppose that the plotting program requires two values with its `new_plot` event, one giving the X coordinates of the points to plot, and the other giving the Y coordinates. This interface is quite natural, but it doesn't fit directly with our `read_BPMs` program, which only generates the BPM readings, not also their numbers. Accommodating this difference is no problem with Glish:

```
whenever bpms->read_done do
    {
    bpm_pos := 1:len($value)
    send plot->new_plot(x=bpm_pos,
                        y=$value)
    }
```

Here we have assigned to the variable `bpm_pos` an array of the integers from `1` to the number of elements in `$value`, i.e., the number of BPM readings.

This example illustrates a crucial difference between Glish and other systems (see [1] for references) for building distributed applications. In systems that only allow directly connecting together different programs, we could not have "plugged" together the `read_BPMs` program and the plotting program without modifying one or the other. With Glish,

however, we can glue together their *different* interfaces using a single extra line in our scripting language. We do *not* have to modify the programs themselves!

We can not only introduce new values when sending out events, but also modify existing values. Suppose, for example, that `read_BPMs` gives the BPM readings in meters but we want to display them in millimeters. We could use:

```
whenever bpms->read_done do
    {
    bpm_pos := 1:len($value)
    send plot->new_plot(x=bpm_pos,
                        y=$value*1000)
    }
```

Glish supports array arithmetic and mixed-type arithmetic. Here, each element of the array `$value` is multiplied by the floating-point value `1000.0` prior to creating the `y` field of the `new_plot` event sent to the plotter.

In general, Glish provides a powerful set of operators for manipulating numeric array data. For example, here is a full implementation of the "quick sort" sorting algorithm, written only using Glish's array-manipulation primitives:

```
func qsort(x)
    {
    if ( len(x) == 0 )
        # already sorted
        return []

    local below := x < x[1]
    local above := x > x[1]
    local same := x == x[1]

    return [qsort(x[below]),
        x[same], qsort(x[above])]
    }
```

Here the local variables `below`, `above`, and `same` are *masks*. That is, they are boolean arrays, each element of which is "true" if the corresponding element of the argument `x` is greater than, less than, or equal to the first element of `x`, and "false" otherwise. These masks are used to index `x` to produce only those elements of `x` satisfying the mask. The elements less than and greater than `x[1]` are then recursively sorted and the entire collection of now-sorted elements are put together into a single array by surrounding them with `[]`'s, and we are done sorting. Finally, because Glish is dynamically typed, this function can be used unmodified to sort boolean, integer, single-precision, double-precision, and string arrays.

In addition to the features described above, the Glish language includes records with arbitrarily-typed fields, mechanisms for request/reply events and for suspending script execution until a particular event arrives, direct connections between clients (see the next section), and ways to integrate plain Unix commands (with no knowledge of Glish) into a Glish script (see § 6).

Finally, the Glish interpreter can run scripts interactively, allowing a user to type in arbitrary Glish commands to modify the behavior of an active script. This facility provides a useful debugging tool.

4 The Flow of Data in a Glish Application

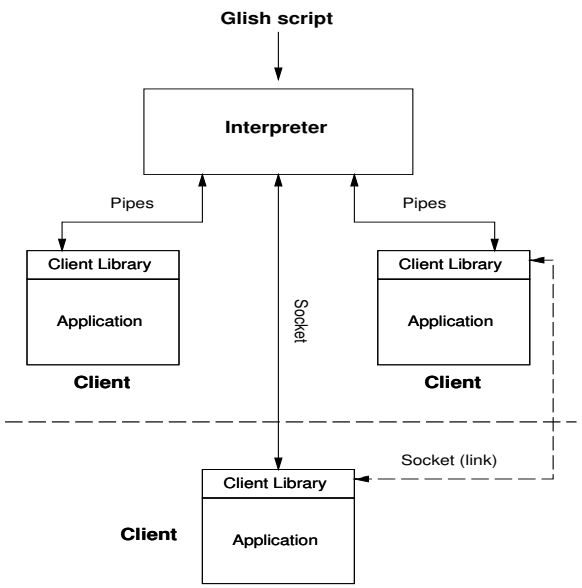

Figure 1: Sample Glish Application Architecture

Glish scripts are run by the Glish *interpreter*, which controls all process creation and communication specified by the script. Figure 1 shows the details of how events flow through a Glish application. First, the Glish script is given to the interpreter, which executes the statements in it. In this example, those statements direct the interpreter to create three clients, two running on the same host and one running on a remote host (shown below the thick dashed line). The interpreter uses Unix pipes to communicate with the first two clients and a TCP socket to communicate with the third. Each program links in the Glish Client library (see § 5), which manages the communication channel.

Events are sent over the sockets or pipes as typed, self-describing binary data. If the two ends of the communication channel support the same binary data formats, no conversion needs to be made of the data; if they are heterogeneous, however, then the data are converted between architecture types, transparent to the programs.

As indicated by the diagram, in general all communication flows from clients to the interpreter and then perhaps from

the interpreter on to one or more other clients. This centralized design gives Glish its main power, that of being able to use Glish's high-level language to modify the data sent between programs without modifying the programs themselves. It also, however, doubles communication costs in the common case of one client forwarding its events directly along to another.

For most purposes, the centralized performance is adequate: on a single unloaded Sparcstation 2, we measured a client-to-client event rate of about 150 empty events/sec, and about 100 events/sec when sending 8Kbyte events. When running between the Sparcstation 2 and a Sun IPC on the same Ethernet, we measured about 135 empty events/sec, and 37 8Kbyte events/sec.

For most high-level control applications, these rates are acceptable. In those cases requiring greater performance, Glish supports establishing "point-to-point links" between clients. Figure 1 shows such a link using a socket between the right-hand and bottom clients. Such links remain transparent to all but the Client library layer of the two clients. Point-to-point links double the effective data rate, but at a cost of losing the opportunity to modify the data sent between clients.

5 Writing Glish Clients

A program's interface to the Glish system comes via linking with the C++ "Client" library. This library exports two main classes, `Client`, for managing the event connection to the Glish interpreter (and to other clients), and `Value`, for encapsulating data values in order to send them over the software bus. You create a Glish client by having your program instantiate a `Client` object, passing it the arguments with which the program was invoked:

```
int main( int argc, char** argv )
    {
    Client c( argc, argv );
    ...
```

You can then either simply enter a loop in which you repeatedly ask the `Client` object for the next event and dispatch based on the event's name, or you can ask it for a file descriptor mask suitable for handing to the Unix *select()* function, which allows your program to multiplex multiple event streams.

Events arrive with their values encapsulated in a single `Value` object. The `Value` class supports numerous methods for accessing and manipulating its contents. You can also in turn create `Value` objects encapsulating the scalar, array, string, and record values used by your program. You send events out into the Glish world using the `PostEvent` method of the `Client` object, passing it the name of the event and a corresponding `Value` object.

Note that while the outermost event-loop of your program must be written in C++, because C++ integrates well with C (and, to a lesser degree, with FORTRAN), you are not confined to writing your entire program in C++.

6 Integrating Existing Programs

One of the goals of Glish is to facilitate the use of existing, non-Glish programs when creating a new Glish application. Glish supports this goal in several ways. First, you can run Unix shell commands from within a Glish script, sending them arbitrary text via *stdin*, waiting for them to complete, and capturing their *stdout* output in a string array. Next, you can run the same sort of commands *asynchronously*, in which case you send them `stdin` events to make text appear on their *stdin*, and receive `stdout` events from them representing their output. Finally, often it is straight-forward to add a C++ "event wrapper" around an existing program's routines, giving the program an event-oriented structure and encapsulating the program's key data structures in `Value` objects. The orbit-correction application discussed in § 8 below used this approach for its interface to the Teapot modeling program (written in C) and the "clorb" orbit-correction algorithm (written in FORTRAN).

7 Dealing With Failure

A key need of any control application is resilience in the presence of failure. Glish provides two event-based mechanisms for detecting and responding to failure.

The first mechanism is that the Glish interpreter detects whenever a Glish client terminates improperly. If a Glish client exits without properly terminating its connection to the Glish interpreter, the interpreter detects this fact (due to the lost pipe or socket connection) and generates a `fail` event on the process' behalf. Thus,

```
whenever my_client->fail do
    oops()
```

can be used to call the `oops()` function whenever `my_client` terminates abnormally.

The second mechanism involves detecting network outages. The Glish interpreter periodically polls the daemons running on its behalf on other network hosts. If it loses contact with the daemon, the predefined `system` client generates a `connection_lost` event. If the connection returns, it generates a `connection_restored` event, and if the daemon crashes, a `daemon_terminated` event. All of these events may be used in `whenever` statements to execute whatever (arbitrary) response is appropriate.

On a more general note, because clients do not depend directly on one another for their communication channels, the failure of one client does not tend to cascade and crash other clients. Instead, clients that received events sent by the

failed client simply sit idle in their event loops (and continue to process events sent to them by other clients). If the Glish script is being run interactively, it may be possible to restart the failed client or begin another in its place and continue.

8 An Example of a Glish Application

In this section we give an overview of how Glish is used for correcting the beam orbit of the Advanced Light Source at LBL. For more details, see [2].

While orbit-correction might sound like a fairly simple application—all one does is get the trajectory, calculate the corrector settings to alter it as desired, and apply those changes—in reality it is quite a complex task. In particular, one must deal with broken, disabled, and untrustworthy BPM's; different hardware used to measure first-turn and closed-orbit trajectories; different algorithms required to correct each of these; the need to sometimes average trajectory readings over multiple turns; known or suspected offsets in the BPM's; varying calibration factors for converting between amperes and milliradians of corrector kick; the need to apply corrections in steps to avoid losing the beam or applying an erroneous correction due to hysteresis effects; sometimes using nominal Twiss parameters while other times wanting to compute them from beam measurements; and needing different "goal" orbits for different correction procedures.

The resulting correction application uses seven different Glish clients—ranging from a graphical user-interface to a FORTRAN correction algorithm to data archiving to accessing the underlying hardware—and 40 different events. It addresses all of the problems discussed in the previous paragraph, and in addition illustrates how Glish aids in overcoming the difficulties of high-level control applications outlined at the beginning of the paper. In particular, none of the constituent clients is hideously large (the user-interface is the largest, at around 3,000 lines of code); a number of preexisting programs were used, written by different people and with different physics conventions and units, some utilizing a separate, existing RPC library; the application runs on-line, in simulation using a modeling program, or off-line on archived data; the application works with both the ALS booster and the storage ring, requiring only switching a configuration file and changing the client used to access the underlying hardware; and the application has proven quickly adaptable to unforeseen problems (for example, one night some of the BPM readings were coming back as bogus values, $\pm 10^{30}$ mm.; this problem was corrected for, without altering the BPM-reading software, with a two-line fix in the Glish script).

9 Summary

The Glish software bus system has matured a great deal from the prototype system described in [3]. It has now been used for control applications at the Advanced Light Source at LBL, and at the Magnet Test Laboratory at SSCL. A number of other sites, both national laboratories and commercial institutes, have copies for experimentation purposes. Finally, Brookhaven National Laboratory plans to incorporate Glish into the control system for RHIC [4].

We feel Glish has proved quite successful at addressing the challenges given in the introduction of this paper. Our experience at the ALS with using Glish in parallel with a separate RPC facility suggests that Glish will readily work in conjunction with other control-system toolkits such as EPICS[5], letting them take care of real-time concerns and itself providing powerful "glue" for rapidly constructing high-level control applications.

The current release of Glish is version 2.4. Source code and full user documentation can be retrieved via anonymous ftp to `ftp.ee.lbl.gov`, in the *glish/* subdirectory. The author can be reached via email to `vern@ee.lbl.gov`.

10 Acknowledgements

I would like to thank my colleagues at LBL, SSCL, and BNL for their input and feedback on the design and use of Glish, especially Chris Saltmarsh, Matt Fryer, Lindsay Schachinger, Mike Allen, Joe Garbarini, and Dave Lambert.

This work was supported by the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

References

[1] V. Paxson and C. Saltmarsh, *Glish: A User-Level Software Bus for Loosely-Coupled Distributed Systems*, Proceedings of the 1993 Winter USENIX Conference, San Diego, CA, January, 1993.

[2] L. Schachinger and V. Paxson, *A Software System for Modeling and Controlling Accelerator Physics Parameters at the Advanced Light Source*, Proceedings of the 1993 IEEE Particle Accelerator Conference, Washington, D.C.

[3] V. Paxson, C. Saltmarsh, M. Allen and M. Kane, *A Language, Server and C++ Class Library for Event Sequencing*, Proceedings of ICALEPCS '89, Nuclear Instruments and Methods in Physics Research, A293, pp. 356-362, 1990.

[4] S. Peggs, C. Saltmarsh, T. Satogata, and M. Fryer, *High Level Controls at RHIC*, submitted to ICALEPCS '93.

[5] L.R. Dalesio, M.R. Kraimer, and A.J. Kozubal, *EPICS Architecture*, Proceedings of ICALEPCS '91, KEK Proceedings 92-15, p. 278, December, 1992.