Appendix: Glish Syntax and Grammar

The Glish syntax is free-form. Comments begin with # and extend to the end of the line. Statements are formally terminated with semi-colons but in general Glish is able to infer the end of a statement and supply an implicit terminator at the end of a line. Identifiers are case-sensitive; record field names and event names have separate name spaces and may include keywords.

In the following grammar, []'s surround optional elements and {}'s surround elements that may occur zero or more times. Terminals are surrounded with quotes or appear in uppercase.

```
program:  { stmt }

stmt:  "{" { stmt } "}"
     |  WHENEVER ev-list DO stmt ";"
     |  LINK ev-list TO ev-list ";"
     |  UNLINK ev-list TO ev-list ";"
     |  AWAIT ev-list ";"
     |  AWAIT ONLY ev-list
              [EXCEPT ev-list] ";"
     |  event "(" [param-list] ")" ";"
     |  IF "(" expr ")" stmt
                      [ELSE stmt]
     |  FOR "(" ID IN expr ")" stmt
     |  WHILE "(" expr ")" stmt
     |  NEXT ";"
     |  BREAK ";"
     |  RETURN [expr] ";"
     |  EXIT [expr] ";"
     |  PRINT [param-list] ";"
     |  LOCAL id-list ";"
     |  expr ":=" expr ";"
     |  expr ";"
     |  ";"


expr:  "(" expr ")"
     |  expr logop expr
     |  expr relop expr
     |  expr arithop expr
     |  expr ":" expr
     |  expr "[" expr "]"
     |  expr "(" [param-list] ")"
     |  expr "." FIELD-ID
     |  unaryop expr
     |  "[" "=" "]"
     |  "[" [param-list] "]"
     |  function
     |  LASTEVENT
     |  ID
     |  CONSTANT

logop:  "|"  |  "||"  |  "&"  |  "&&"
relop: "=="  |  "!="  |  "<"  |  "<="
     |  ">"  |  ">="
arithop: "+"  |  "-"  |  "*"  |  "/"
      |  "%"  |  "^"
unaryop: "-"  |  "+"  |  "!"  |  ref-type

function: func-head "(" [formal-list] ")"
          func-body

func-head: FUNCTION [ID]
    |   SUBSEQUENCE [ID]

func-body: "{" { stmt } "}"
    |  expr

formal: [ref-type] ID ["=" expr]
    |   "..."

ref-type:  VAL  |  REF  |  CONST

param: expr
    |  ID "=" expr
    |  "..."

event: expr "->" EVENT-ID
    |  expr "->" "[" expr "]"
    |  expr "->" "*"

ev-list:  event ["," ev-list]
id-list:  ID ["," id-list]
param-list:  param ["," param-list]
formal-list:  formal ["," formal-list]
```

ing the system design, our experiences to date have convinced us that with our software bus "shell" we have in place a firm foundation for building distributed applications.

11 Acknowledgements

We would like to thank our *ISTK* collaborators for their input and feedback on the design and use of Glish, especially Matt Fryer, Matthew Kane, and Mike Allen.

We would also like to thank Steve McCanne, Lindsay Schachinger, Matt Fryer, and the referees for their many helpful comments on various drafts of this paper.

For the curious, the Glish language was named by the second author so that when upper management types ask, "What language is this stuff written in?" we can reply, "*In Glish*, of course!"

References

[BCW88] Richard A. Becker, John M. Chambers and Allan R. Wilks, "The New S Language", Wadsworth & Brooks, Pacific Grove, CA, 1988.

[Cagan90] Martin R. Cagan, *The HP SoftBench Environment: An Architecture for a New Generation of Software Tools*, Hewlett-Packard Journal, 41(3), pp. 36-47, June, 1990.

[CG89] Nicholas Carriero and David Gelernter, *Linda in Context*, Communications of the ACM, 32(4), pp. 444-458, April, 1989.

[Fromme90] Brian D. Fromme, *HP Encapsulator: Bridging the Generation Gap*, Hewlett-Packard Journal, 41(3), pp. 59-68, June, 1990.

[HGJMLR89] D. E. Hall, W. H. Greiman, W. F. Johnston, A. X. Merola, S. C. Loken and D. W. Robertson, "The Software Bus: A Vision for Scientific Software Development", Proceedings of the International Conference on Computing in High Energy Physics, Oxford, England, 1989.

[KMN89] Jeff Kramer, Jeff Magee and Keng Ng, *Graphical Configuration Programming*, IEEE Computer, 22(10), pp. 53-65, October, 1989.

[MKS89] Jeff Magee, Jeff Kramer and Morris Sloman, *Constructing Distributed Systems in Conic*, IEEE Transactions on Software Engineering, 15(6), pp. 663-675, June, 1989.

[Ouster90] John K. Ousterhout, "Tcl: An Embeddable Command Language", Proceedings of the 1990 USENIX Winter Conference, Washington, D.C., January, 1990.

[PSAK90] V. Paxson, C. Saltmarsh, M. Allen and M. Kane, *A Language, Server and C++ Class Library for Event Sequencing*, Nuclear Instruments and Methods in Physics Research, A293, pp. 356-362, 1990.

[Reiss90] Steven P. Reiss, *Connecting Tools Using Message Passing in the Field Environment*, IEEE Software, 7(4), pp. 57-66, July, 1990.

[Scott87] Michael L. Scott, *Language Support for Loosely Coupled Distributed Programs*, IEEE Transactions on Software Engineering, 13(1), pp. 88-103, January, 1987.

[Skeen92] Dale Skeen, "An Information Bus Architecture for Large-Scale, Decision-Support Environments", Proceedings of the 1992 USENIX Winter Conference, San Francisco, CA, January, 1992.

[UCAR91] University Corporation for Atmospheric Research, "NetCDF User's Guide", available via anonymous `ftp`; retrieve pub/netcdf.tar.Z from host `unidata.ucar.edu`.

[WS91] Larry Wall and Randal Schwartz, "Programming Perl", O'Reilly & Associates, Sebastopol, CA, 1991.

Author Information

Vern Paxson holds an M.S. degree in computer science from U.C. Berkeley. He has been on the staff at the Lawrence Berkeley Laboratory since 1985, primarily working on software for accelerator physics simulation and control. Since 1991 he has also been a Ph.D. student at U.C.B. in the area of wide-area networking. Vern's claim to net-fame is as the author of *flex*, a high-performance *lex* rewrite. Reach him at `vern@ee.lbl.gov`.

Chris Saltmarsh obtained his doctorate at Nottingham University in cosmic ray physics, and then worked at CERN, firstly on the NA7 pion form factor experiment and then in the operations and machine physics groups of the Super Proton Synchrotron. He moved to the U.S. some 5 years ago to work with the Central Design Group of the Superconducting Super Collider and is presently with the SSCL working from the Lawrence Berkeley Laboratory. Reach him at `salty@largo.lbl.gov`.

Both authors can be reached via U.S. mail at Lawrence Berkeley Laboratory, MS 46-A/1123, 1 Cyclotron Rd., Berkeley, CA 94720.

client connected via Ethernet, we found real-times (excluding startup overhead for running the remote daemon) averaging 7.3 seconds for empty events and 26.2 seconds for 8KB-events. These values correspond to about 275 empty events/sec, 75 8KB-events/sec, and a data rate of 300 KB/sec, about 25% of the raw Ethernet bandwidth.

Note that these rates correspond to the performance available when using point-to-point links. Forwarding events via the Glish interpreter halves the rates.

8 Related Work

It is widely held ([Scott87, CG89, MKS89, Cagan90, Reiss90]) that to build flexible distributed systems the individual programs in the system should have no knowledge of the inter-program connections (i.e., where their input comes from and where their output goes to). Extending this notion with self-describing data to form a "software bus" is discussed in [HGJMLR89] and [Skeen92], the former a "concept" paper and the latter a description of a proprietary system.

Many approaches to building distributed systems rely on special operating system support or writing client programs in specialized languages. For our purposes it was important that the system be portable between different Unix systems without kernel modifications, and that we be able to incorporate into the system existing programs written in C, C++, or FORTRAN.

Four systems that work with little operating system support and can integrate existing programs are Tcl [Ouster90], HP Softbench [Cagan90], Linda [CG89], and Field [Reiss90]. Tcl and Field limit interprocess communication to strings, making efficient communication of binary data problematic. Tcl also does not provide mechanisms for starting new processes. HP Softbench communicates data via the file system, requiring operating system support for network communication. (The related HP Encapsulator [Fromme90], though, provides a nice way to integrate existing programs into the system.)

The essence of all of these systems is enabling event "producers" and event "consumers" to find one another and communicate. Glish's main contribution is that it also provides a powerful language for manipulating both interprocess connections and the contents of the data passed between programs. In this respect Glish makes it easy to integrate programs with *different* interfaces; Glish provides the "glue" to bridge the differences between what one program generates and what another program expects.

9 Present Status

What we have described is the third generation of Glish (the first, very different implementation is discussed in [PSAK90]). We found that redesigning the language twice, while painful, greatly clarified and enriched it in result. Glish now has the flexibility to accommodate our simulation and control applications. Glish is used to control testing of superconducting magnets at the Superconducting Super Collider Laboratory (SSCL) and for analysis, simulation, and control of the Advanced Light Source accelerator at the Lawrence Berkeley Laboratory (LBL). Its use has been growing and we anticipate continued evolution of the system.

The current release of Glish is version 2.1. Source code can be retrieved via anonymous ftp to `ftp.ee.lbl.gov`. The current release includes all of the features described above except that client event values whose types are functions, agents, or references are not yet supported. The release includes the *SDS* layer used for communication between heterogeneous architectures. The *SDS* code is quite old and never made it out of prototype because of its success; we are rewriting it.

Glish is part of *ISTK* (Integrated Scientific Toolkit), a software package developed primarily by SSCL and LBL. *ISTK* includes a class library for creating objects that automatically change their value when Glish events are received, and send out new Glish events when their value are changed by other means (such as a user-interface). *ISTK* also includes a corresponding graphics library for building user-interfaces, making it simple to tie buttons to arbitrary multiprocess actions, or to automatically update displays when the data they reflect has been altered by another Glish client (or the Glish script itself). Along with these libraries *ISTK* also includes Glish-related applications such as a program for multiplexing string-valued Glish events and keyboard input into terminal-based programs, and a program for displaying Glish events as they are sent between clients. *ISTK* is not yet ready for general release, though interested parties may contact the second author for further information.

10 Future Work

A powerful addition to Glish would be having Glish clients "register" the events they respond to along with type signatures for those events, similar to the use of message patterns in Field, HP SoftBench, and Linda. Glish could then automatically connect together clients with similar event patterns, providing any necessary glue for accommodating differences. By including a "help" string with each registered event, the system could also interactively give the user help and type information on all events generated by all available clients. It then becomes possible to write a visual interface for composing Glish scripts, similar to that for Conic [KMN89].

Other areas to explore are using shared memory for same-host communication (the *SDS* layer already supports this), out-of-band and prioritized events, richer exception handling than just `fail` events, and mechanisms for connecting multiple Glish interpreters together via events.

While we need more experience using Glish before finaliz-

```
#include <string.h>
#include "Glish/Client.h"

// Computes the FFT of the first "len" elements of "in", returning
// the real part in "real" and the imaginary part in "imag".
extern void fft( double* in, int len, double* real, double* imag );

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

        GlishEvent* e;
        while ( (e = c.NextEvent()) )
                {
                if ( ! strcmp( e->name, "fft" ) )
                        { // an "fft" event
                        Value* val = e->value;

                        // Make sure the value's type is "double".
                        val->Polymorph( TYPE_DOUBLE );
                        int num = val->Length();

                        // Get a pointer to the individual elements.
                        double* elements = val->DoublePtr();

                        // Create arrays for results.
                        double* real = new double[num];
                        double* imag = new double[num];

                        // Compute the FFT.
                        fft( elements, num, real, imag );

                        // Create a record for returning the
                        // two arrays.
                        Value* r = create_record();
                        r->SetField( "real", real, num );
                        r->SetField( "imag", imag, num );

                        c.PostEvent( "answer", r );
                        Unref( r );
                        }
                else
                        c.Unrecognized();
                }
        return 0;
        }
```

Figure 5: Glish Wrapper for *FFT* Client

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

The *Client* class provides three main member functions:

- `NextEvent` waits for the next event to arrive and returns its name and a corresponding *Value* object. The event is returned as a pointer to a *GlishEvent* object, which is simply a structure with `name` and `value` fields.
- `PostEvent` takes a string and a *Value* object and sends an event with the given name and value.
- `Unrecognized` is used to report that the current event is not recognized by the Glish client.

The class also provides variants on `PostEvent` for sending events with simple string values. In addition, the class provides access to the file descriptors from which it reads events, so the program can use *select* to multiplex between different input sources.

If the program was *not* invoked by the Glish interpreter then the special arguments will be missing. The *Client* library detects this case and knows that the program is running standalone, in which case it reads string-valued events from *stdin* and "posts" outbound events to *stdout*. This behavior allows client programs to be debugged separate from running within Glish.

5.3 An Example of a Client

Suppose we want to create an "FFT server": a Glish client that when sent a numerically-valued `fft` event computes the FFT of the array of data and returns the result as an `answer` event. The result consists of a record with two fields, `real` and `imag`, arrays of the real and imaginary parts of the Fourier transform.

Assume we have a function *fft* available for doing the actual transformation and want to "wrap" a Glish client interface around this function. Figure 5.3 shows how we would do so.

First we create a *Client* object using the idiom discussed in Section 5.2. We then enter the event-loop, blocking until a new event is ready (`NextEvent` returns a nil pointer when the client should terminate).

If the event's name is `fft` then we extract the event's value, convert it to "double" if it is not already, and extract its length into *num*. We then use `DoublePtr` to get a pointer to the actual array of double-precision elements. In order to call *fft* we need to also pass it arrays where it should put its results, so we create *real* and *imag*. After computing the FFT we create in *r* a Glish record value to hold the two arrays, and assign them to *r*'s `real` and `imag` fields. We then send this aggregate value as a Glish event with the name `answer`. Now that we're done with *r* we `Unref` it to reclaim its memory. This will automatically result in *real* and *imag*'s memory being reclaimed too. We don't need to `Unref` the *GlishEvent* pointed to by `e` because the next call to `NextEvent` automatically does so.

Finally, if the event wasn't `fft` then we inform the *Client* library that we don't recognize this particular event.

6 Implementation

The Glish language is implemented as an interpreter, written in about 10,000 lines of C++. It runs on SunOS, Ultrix, and HP/UX platforms; Glish clients can also run on VxWorks, using a limited client library written in C instead of C++.

We chose an interpreter implementation because it gives very fast turn-around times when modifying scripts, as well as the ability to run interactively. The interpreter is optimized to perform array-wise operations in tight loops, making its overhead acceptable.

In general inter-client communication goes through the interpreter; the design is *centralized*, much like the designs of Field [Reiss90] and HP SoftBench [Cagan90]. Remote communication occurs via TCP sockets, assuring reliable delivery of events, while local communication uses pipes for added performance. Point-to-point links enable faster but less flexible communication. We implemented them using named pipes for same-host communication and sockets for remote communication.

To create clients on a remote host the interpreter first remotely executes a daemon on that host to execute and control processes on its behalf, much like the SPC daemon used by HP SoftBench. All event communication with remote clients is still done directly between the client and the interpreter via a socket connection.

Event values are sent using a self-describing dataset format called *SDS*, similar to netCDF [UCAR91]. *SDS* handles padding, byte-swapping, and floating-point representation differences, so it can be used to efficiently transmit binary data between heterogeneous architectures (e.g., VAX and SPARC). The *SDS* layer is written in C (about 9,000 lines).

7 Performance

To give a feel for Glish's performance, on an unloaded Sparcstation 2 sending an empty event back and forth to a "ping" client on the same machine for 1,000 round trips takes an average of 6.5 real-time seconds, for an event rate of around 300 events per second. Sending 8KB-events takes an average of 10.3 real-time seconds, for a rate of about 200/sec and a data rate of 800 KB/sec. The CPU times (user + system) were about 60% of the real-time timings.

When making the same timings between a Sparcstation 2 running the Glish interpreter and a Sun IPC running the "ping"

```
subsequence power(exponent)
    {
    whenever self->compute do
        self->ready( $value ^ exponent )
    }

square := power(2)
cube := power(3)

square->compute( 6 )
cube->compute( [2, 5, 10.1] )

whenever square->ready, cube->ready do
    print $value
```

Figure 4: Example of a Glish Subsequence

The final way to create an agent is using a *subsequence*. A subsequence is just like a function except that when called it returns an agent value, which can be used to send and receive events to and from the subsequence. In the body of a subsequence the predefined variable `self` refers to its agent value. For example, the script shown in Figure 4 creates two subsequences. When executed it prints `36` followed by `[8 125 1030.3]`.

The first set of statements defines `power` as a subsequence that is invoked with an argument `exponent` and responds to `compute` events by generating a `ready` event whose value is the value of the `compute` event raised to the given exponent. The two assignments bind `square` and `cube` to agents corresponding to different instances of `power`. The next two statements send those agents `compute` events with a single integer value and a three-element double-precision array value, respectively. The final `whenever` statement prints the value of any `ready` events generated by `square` or `cube`.

5 The Client Library

Programs interface to the Glish system via the Glish "Client" library, which is written in C++. The library exports two classes: *Value* and *Client*. *Value* objects correspond with Glish values: they are dynamically typed arrays, records, functions, or agents. The *Client* class provides the mechanism for a Glish client to send and receive events.

5.1 The *Value* Class

Value objects can be constructed from C++ scalars or arrays. For example,

```
Value* v = new Value( 5 );
```

assigns to `v` a *Value* object representing the integer 5, while

```
double* x = new double[3];
x[0] = 1.0;
x[1] = 3.14;
x[2] = 4.56;
Value* v = new Value( x, 3 );
```

assigns to `v` the equivalent of the Glish value `[1, 3.14, 4.56]`. By default, *Value* objects constructed from arrays "take over" the array: they will *realloc* the array if it grows larger and delete it when the *Value* object is destroyed. The class library also provides mechanisms for specifying that an array should not be altered or should first be copied.

The *Value* class provides a number of member functions for manipulating values:

- `Type` returns the type of an object and `Length` its length.
- `IntVal` interprets one element of the value as a single integer, performing coercions as necessary, and similar functions are provided for boolean, floating-point, and string interpretations.
- `IntPtr` returns a pointer to a C++ array of integers that can then be used for direct access to the value's underlying elements, while `CoerceToIntArray` returns either the underlying array if already of type *integer* or else a copy of the array converted to *integer*. Again, these functions have counterparts for the other Glish types.
- `Polymorph` converts the value from its present type to a new type.
- Analogs to these functions are available for directly accessing and setting a record's fields.
- The (non-member) function `create_record` returns a new, empty record.

A key point concerning the *Value* class is that it makes it easy to wrap Glish values around an existing program's data structures. These data structures can then be made available to other programs by sending them as event values.

Note also that both the *Value* and *Client* classes use reference-counting for memory management. The `Ref` and `Unref` functions manipulate each object's reference count. When the count reaches zero the object is deleted and any objects it refers to are `Unref`'d.

5.2 The *Client* Class

Each Glish client constructs one instance of the *Client* class by passing the *Client* constructor the program's `argc` and `argv`. When a Glish client is executed by a Glish script `argv` contains special arguments telling the *Client* object how to connect the Glish interpreter. So usually the beginning of a Glish client looks like:

After an `await`, `$agent`, `$name`, and `$value` correspond to the event that caused the `await` to finish. In the above example, `$agent` will be `c`, `$name` will be `"compute_done"`, and `$value` will be whatever value the `compute_done` event had.

Any other events that arrive during an `await` are still processed by Glish (i.e., it executes the body of any corresponding `whenever` statements). An `await only` statement can be used to tell Glish to drop these events instead. It is meant for use as a "hold-point", to freeze the effective execution of a Glish script until some seminal event occurs. Glish also provides a mechanism for listing exceptions to this rule, so that certain high-priority events will still be processed during an `await only`.

4.6 Point-to-Point Communication

Sometimes in a Glish system two clients need to communicate as fast as possible. If the system's Glish script only forwards events from one client to the other without modifying the events' values then we can instead use a direct connection between the two. Glish supports this style of communication using the `link` statement. When executed a `link` statement directs a client to send a particular event it generates directly to another client (perhaps renaming it). For example,

```
link t->transformed_data to
     d->new_data
```

will cause the client associated with `t` to send its `transformed_data` events directly to `d`'s client, which will see them as `new_data` events. (Other events generated by `t`'s client still go to the Glish interpreter.) The destination of a link can use the "`*`" event to mean "use the same name":

```
link t->transformed_data to d->*
```

will send the `transformed_data` events along without renaming them.

You can suspend point-to-point links with the `unlink` statement:

```
unlink t->transformed_data to
       d->new_data
```

suspends the link formed in the first example above. `t`'s agent will now instead send its `transformed_data` events to the Glish interpreter, which will execute the corresponding `whenever` bodies. Executing another `link` statement restores the point-to-point link.

4.7 Creating Agents

Agent values can be created three different ways. First, the predefined function `client` takes an *argv*-style list of strings and instantiates the corresponding program with the given arguments. `client` also has optional arguments for specifying on which host to run the process and whether to initially suspend the process to allow a debugger to be attached. For example,

```
t := client("timer", 5, host="psychosis")
```

runs the Glish client *timer* on the remote host "psychosis" with an argument of 5 (for "timer" this is the timer interval in seconds) and assigns to `t` an agent value corresponding to this process.

Another way to create an agent is to use the `shell` function with the optional argument `async=T` (`T` is the boolean "true" constant). Asynchronous shell clients can be sent `stdin` events to make text appear on their standard input, `EOF` events to close their standard input, and `terminate` events to terminate them. Each line of text they write to their standard output becomes a `stdout` event.

For example, here's a Glish script that uses *awk* to print the numbers from 1 to 32 in hexadecimal, each appearing as a separate event:

```
cvt := "awk '{ printf(\"%x\\n\", $1) }'"
hex := shell( cvt, async=T )

count := 1
hex->stdin(count)

whenever hex->stdout do
    {
    print count, "=", $value
    if ( count < 32 )
        {
        count := count + 1
        hex->stdin(count)
        }
    else
        hex->EOF()
    }
```

The first two statements associate an asynchronous shell client with the variable `hex`. The next line initializes the global `count` to 1 and sends that value to `hex`, making it appear on *awk*'s standard input.

The `whenever` body prints out the current count and its hexadecimal equivalent, and then either increments the count and sends *awk* a new input line or closes its standard input. Because Glish uses pseudo-ttys to communicate with asynchronous shell clients, *awk*'s output will be line-buffered, so each `stdin` event will shortly result in a new `stdout` event.

One might think that a race exists between sending the first `stdin` event to `hex`'s client and setting up the `whenever` to deal with the client's response. This problem does not arise, however, because the Glish interpreter does not read events generated by clients until it is done executing all of the statements in a script.

```
a->foo( "value1", 2 )
```

sends an event with two values, the string `"value1"` and the integer `2`. The values can also be named:

```
a->foo( x="xval", y=5 )
```

sends an event with the "parameter" `x` equal to `"xval"` and `y` equal to `5`. Multi-valued events are equivalent to passing a single-valued event where the value is a record. This last example is equivalent to:

```
a->foo( [x="xval", y=5] )
```

The event name in a `->` operation needn't be fixed in advanced. Instead you can use any string-valued expression by enclosing it within brackets (`[]`'s). The following are equivalent:

```
a->foo( 5 )
a->["foo"]( 5 )
```

and here is one way to send `a` three events, `foo`, `bar` and `bletch`, with values of `1`, `2`, and `3`:

```
for ( i in 1:3 )
    a->["foo bar bletch"[i]]( i )
```

(Recall that `"foo bar bletch"` is a three-element array of strings.)

One major difference between sending an event and calling a function is that sending an event is an *asynchronous* operation. As soon as Glish has sent the event it proceeds to execute the next statement in the Glish script. Events can be sent synchronously using the `await` statement, which we discuss in Section 4.5 below.

4.4 Receiving Events from Agents

Again, suppose that `a` is a variable with an *agent* value. In a Glish program you can respond to events that `a` generates using a `whenever` statement. Once executed,

```
a := client("demo")
whenever a->bar do
    print "got a bar event"
```

will print `"got a bar event"` every time *demo* generates a `bar` event.

The value of the most recently received event is kept in a special variable `$value`:

```
whenever a->bar do
    print "got a bar event =", $value
```

will display the value of each `bar` event that `a` generates.

Agent values are also records, and the most recent value of each event is available as a field in the record. For example, the `print` statement in the `whenever` above could also have been written:

```
print "got a bar event =", a.bar
```

The value persists in `a.bar` until `a` generates another `bar` event, at which point `a.bar` is updated to reflect the new event's value.

Just as when sending events you can use a string-valued expression to name an event, so can you with `whenever`:

```
whenever a->["foo bar bletch"] do
    print $value
```

will print the value of each `foo`, `bar`, and `bletch` event generated by the agent `a`.

Finally, "`*`" can be used to indicate *every* event:

```
whenever a->* do
    print $value
```

prints the value of every event `a` generates.

Along with `$value`, two other special variables are available in the body of a `whenever`: `$name` holds the name of the event and `$agent` is a reference to the agent that generated it. For example, the following function:

```
function setup_relay(src, ref dest)
    {
    whenever src->* do
        dest->[$name]($value)
    }
```

executes a `whenever` statement relaying every event generated by the agent `src` to the agent `dest`. (`dest` has to be declared a `ref` parameter since sending an event to an agent is considered to modify the agent.) Note that the `whenever` statement "persists" even after a call to `setup_relay` returns.

There is no restriction on the body of a `whenever`. It can include function calls, agent creation, and further `whenever` statements, for example.

4.5 Receiving Events Synchronously

An `await` statement instructs Glish to wait for an event to occur. Glish pauses program execution until this happens. For example, suppose that `c` refers to a client that when sent a `compute` request performs some computation and generates a `compute_done` event when finished. If after you tell `c`'s client to do its computation you want to wait for the result, you could use:

```
c->compute()
await c->compute_done

# at this point, c is done
# with its computation
```

then when `diff` is called with only one argument `b` will be set to 1. So the call `diff(3, 7)` returns `-4`, and the call `diff(3)` returns `2`.

In a function call you can also give the function arguments by name instead of positionally:

```
diff(b=4, a=7)
```

returns 3, since $7 - 4 = 3$.

The function definition above assigns a function value to the global variable `diff`. Functions can also be assigned to local variables and record fields:

```
data.transform :=
    function(x) log(x)/log(2)
```

assigns to the `data` record's `transform` field a function that returns $\log_2$ of its argument.

Arguments to Glish functions can be passed by value, by reference, or by `const` reference (the default), by preceding the argument's name in the function definition with `val`, `ref`, or `const`. Passing by reference allows Glish functions to deal with large values efficiently. Glish also supports variable argument lists, which are useful for writing "wrapper" functions that call other functions. For example,

```
function psych_client(...)
    client(..., host="psychosis")
```

defines a function that when called creates a Glish client on the remote host "psychosis".

One particularly useful predefined function is `shell`, which interprets its arguments as a Bourne shell command line and returns the output from running the command (optionally on a remote host) as a string value. For example,

```
csh_man := shell( "man csh" )
```

assigns to the variable `csh_man` a string array, each element corresponding to one line of the "csh" manual page, and

```
function to_lower(x)
    shell("tr A-Z a-z", input=x,
          host="cruncher")
```

returns its argument converted to lower-case, doing the work on the remote host "cruncher".

The `function` keyword can be abbreviated as `func`.

4 Events and Agents

Glish's main purpose is to coordinate a number of processes that form a distributed system. These processes are instances of programs written in compiled languages such as C or C++.

Each program is written in an *event-oriented* style; the program's sole view of the rest of the system comes from the *events* it receives, and its sole mechanism for communicating its state and results to the system is by generating more events. The programs have no knowledge of what other programs the system includes, or what is done with their results, or where received events came from. The *event-oriented* style lends itself to creating modular programs that you can connect together in novel ways. You make these connections using Glish.

We deal with the details of how programs themselves receive, interpret, and generate events later in Section 5. Here we focus on manipulating events from within a Glish program.

4.1 What is an "Event"?

An *event* has a *name* and an associated *value*. The name is simply an identifier, much like a variable's name. The value can be any Glish value, of any type: numeric, string, record, reference, agent, or function. We might speak of "a `foo` event with value `[3, 2, 5]`" to mean an event whose name is "foo" and value the three-element integer array `[3, 2, 5]`.

4.2 Agents

An *agent* is an entity that generates and responds to events. Typically it's a process running either locally or on a remote computer; these agents are called *clients*.

Agents generate events in order to communicate with the rest of the world, namely the Glish script and any other agents the script may have created. By saying that agents *respond* to events we mean that they expect to receive events with certain names, and when they do they perform some action based on the name and value of the event. The action may entail generating one or more new events or may not.

Glish predefines several events for every agent: `established` is generated when an agent first begins running; `unrecognized` is generated when an agent does not recognize an event sent to it; `done` is generated when the agent finishes successfully; `fail` is generated on behalf of an agent that terminates abnormally (e.g., due to a bus error); and `terminate` can be sent to any agent to tell it to exit. These events form the mechanism by which agents are controlled and errors detected.

4.3 Sending Events to Agents

Suppose that `a` is a Glish variable with an *agent* value. You can send an event to `a`'s agent using the `->` operator. Executing:

```
a := client("demo")
a->foo( [1, 4, 6] )
```

first associates `a` with an instance of the program *demo* running on the local host, and then sends a `foo` event to `a`'s agent (i.e., *demo*) with a value of `[1, 4, 6]`.

Sending an event is in some ways similar to making a function call. In particular, we can send more than one value:

3.3 Records

You can package together a collection of values into a *record*:

```
r := [a="hello", b=11:20]
```

assigns to `r` a record with two fields: `r.a` designates the scalar string `"hello"`, while `r.b` designates a ten-element integer array. You can also create records by directly assigning to a field:

```
s.constants := [3.14159, 2.71828]
```

creates a new record `s` and initializes its `constants` field to an array of two double-precision values.

Besides using the "`.`" operator, you can also access record fields using string-valued array indices:

```
print s["constants"][2]
```

prints `2.71828`. Record fields can also be referred to using integer indices:

```
print r[2]
```

prints the integers from 11 to 20.

3.4 Multi-Element Indexing

Glish provides ways for accessing or modifying more than one array element (or record field) at a time. For example, you can use an integer array as an index into another array:

```
a := [9, -3, 0, 7, 5]
b := [4,2]
print a[b]
```

prints `[7, -3]`. Since the "`:`" operator yields an integer array, you can use it to access a contiguous sequence of elements in an array: `a[3:5]` yields `[0, 7, 5]`.

You can use a boolean array as a *mask* for selecting which elements you want from the array:

```
print x[x >= 4 & x <= 12]
```

prints all the elements of `x` with values between 4 and 12.

Both integer and boolean array indices can also be used to assign to a subset of an array's elements:

```
x[x < 0] := -x[x < 0]
```

negates all of the negative elements in `x`, and

```
rev_x := x[len(x):1]
```

creates in `rev_x` a copy of `x` with the elements in reverse order.

You can select a subset of a record's field in a similar fashion:

```
r := [a=1, b="hi", c=9.3]
s := r["b c"]
```

assigns to `s` a record with two fields, the `b` and `c` fields of `r`.

3.5 References

A reference is a mechanism for two variables to share the same storage for their values. References are created using the `ref` or `const` operators. You can use `ref` references to both access and modify the variable; with `const` references you can only access the variable.

For example,

```
a := 1:5
b := ref a
b[2] := 9
print a
```

prints `[1 9 3 4 5]`.

An important point, though, is that while `a` and `b` refer to the same underlying storage, assigning either of them to another value breaks the connection between the two. If we do:

```
a := 1:5
```

then `a` will go back to equaling `[1 2 3 4 5]` while `b` will remain equal to `[1 9 3 4 5]`.

The reference connection can be maintained, however, by explicitly stating that you want to do so by using the `val` operator. For example, after executing:

```
c := [1, 3, 7, 12]
d := ref c
val c := "hello there"
```

the value of `d` (and of course `c`) will be the two-element string `"hello there"`.

3.6 Functions

Glish provides a flexible mechanism for defining and calling functions. These functions are a data type; they can be assigned to variables or record fields, passed as arguments to other functions, and returned as results of functions. A function body can be either an expression or a block of statements. Here's a simple example of a function that prints its arguments and then returns their difference:

```
function diff(a, b)
    {
    print "a =", a
    print "b =", b
    return a - b
    }
```

You can make arguments optional by specifying default values for them. If in the above example we replaced the first line with:

```
function diff(a, b=1)
```

and pass it along to *display*.

These examples illustrate the main goals of Glish: making it easy to dynamically connect together processes in a distributed system, and providing powerful ways to manipulate the data sent between the processes. One other important point is that because *measure*, *transform*, and *display* are all written in an event-driven style, each of them can be easily replaced by a different program that has the same "event interface". For our own work (scientific programming) we often want to replace *measure* with *simulate* (a program that simulates the quantity being measured), *display* with a non-interactive program once we have ironed out the measurement cycle, and *transform* with a variety of different transformations. We also might want to run *measure* and *simulate* together, so we can compare *simulate*'s model with the actual phenomenon measured by *measure*. The ability to quickly "plug in" different programs in this fashion is one of Glish's main benefits.

3 The Glish Language

3.1 Overview

The design of the Glish language was heavily influenced by the *S* language [BCW88]. Every value is a dynamically-typed array. The *S* types included are *numerics* (boolean, integer, float, and double, all of which can be freely mixed and coerced to one another), strings, functions, and records. Record fields can be accessed using string-valued expressions as well as with the field-name operator, so records provide a form of associative array.

We added two more types: *references* to other values (for efficiently dealing with large arrays) and *agents*, which are event producer/consumers. Agents typically are programs that have been linked with the Glish Client library (in which case they are called *clients*). They can also be shell commands or Glish "subsequences", similar to Glish functions.

Two levels of scoping are provided for variables, global and local to a function. Variables needn't be declared, except to explicitly set their scope. There is no "main" function; statements outside the scope of any function are executed when the Glish script begins. Here, for example, is the Glish "hello, world" program:

```
print "hello, world"
```

The usual control constructs are provided, along with five additional types of statements:

- *event-send* statements for sending events;
- `whenever` statements for specifying what should happen when an event is generated;
- `await` statements for synchronous communication;
- `link` statements for creating point-to-point communication links;
- `unlink` statements for suspending point-to-point links.

These are discussed in Section 4 below.

Glish also provides a number of predefined functions (such as `sqrt`, `max`, `sum`, all array-oriented) and variables. Examples of predefined variables are `argv`, the argument list with which the script was run, and `environ`, a record of the environment variables. For example, the current user name can be accessed using `environ["USER"]` or `environ.USER`.

3.2 Arrays

Most Glish types correspond to an *array* of values rather than a single value. For example,

```
a := [1, 2, 6]
b := [3, 4, 5]
print a + b
```

assigns two three-element integer arrays to `a` and `b`, and then prints their element-wise sum: `[4, 6, 11]`. You can also mix arrays and *scalars* (single-element arrays) in expressions:

```
print a * 2
```

will print `[2, 4, 12]`. Glish provides the usual arithmetic and logical operators; all operate element-by-element on two arrays of the same size, or, if one of the operands is a scalar, apply the scalar value to each element in turn.

Arrays automatically grow when you assign to an element beyond their current end. Given `a` as above, executing:

```
a[5] := 4
```

results in `a` having the value `[1, 2, 6, 0, 4]`.

Integer arrays can also be created using the built-in "`:`" operator, which returns an array of the integers between its operands. For example,

```
3:7
```

is equivalent to

```
[3, 4, 5, 6, 7]
```

String arrays can be created by enclosing text within double quotes. The text is broken into words at each occurrence of whitespace, which is then discarded:

```
c := "hello, world"
```

assigns to `c` a two-element string array, the first element being "`hello,`" and the second element "`world`". Text enclosed in single quotes is treated as a string scalar:

```
d := 'hello, world'
```

assigns to `d` a single-element string value, with the whitespace preserved.

The `length` function returns the length of an array. It can be abbreviated as `len`.

```
    else
        d->new_data( $value )
    }

whenever t->transformed_data do
    d->new_data( $value )

whenever d->take_data do
    m->take_data( $value )

whenever d->set_transform do
    do_transform := $value
```

We initialize `do_transform` to `T`, the boolean "true" constant. We change it whenever *display* generates a `set_transform` event (see the last two lines). When *measure* generates a `new_data` event we test the variable to determine whether to pass the event's value along to *transform* or directly to *display*.

Furthermore, if the data transformation done by *transform* is fairly simple, we could skip writing a program to do the work and instead just use Glish. For example, suppose the transformation is to find all of the `x` measurements that are larger than some threshold, and then to set those `x` measurements to the threshold value and the corresponding `y` measurements to 0. We could do the transformation in Glish using:

```
m := client("measure", host="mon")
d := client("display")
do_transform := T

if ( len(argv) > 0 )
    thresh := as_double(argv[1])
else
    thresh := 1e6

whenever m->new_data do
    {
    if ( do_transform )
        {
        too_big := $value.x > thresh
        $value.x[too_big] := thresh
        $value.y[too_big] := 0
        }

    d->new_data( $value )
    }

whenever d->take_data do
    m->take_data( $value )

whenever d->set_transform do
    do_transform := $value
```

Here we first check to see whether any arguments were passed to the Glish script and if so we initialize `thresh` to be the first argument interpreted as a double precision value. If no arguments were given then we use a default value of one million.

Now whenever *measure* generates a `new_data` event and we want to do the transformation, we set `too_big` to a boolean mask selecting those `x` elements that were larger than `thresh`. We then set those `x` elements to the threshold, zero the corresponding `y` elements, and pass the result to *display* as a `new_data` event. We have eliminated the need for *transform*.

Finally, for situations in which performance is vital Glish provides point-to-point links between programs. The `link` statement connects events generated by one program directly to another program. The `unlink` statement suspends such a link (further events are sent to the central Glish interpreter) until another `link`. Here is the last example written to use point-to-point links:

```
m := client("measure", host="mon")
d := client("display")

link m->new_data to d->new_data

if ( len(argv) > 0 )
    thresh := as_double(argv[1])
else
    thresh := 1e6

whenever m->new_data do
    {
    too_big := $value.x > thresh
    $value.x[too_big] := thresh
    $value.y[too_big] := 0
    d->new_data( $value )
    }

whenever d->take_data do
    m->take_data( $value )

whenever d->set_transform do
  {
  if ( $value )
    unlink m->new_data to d->new_data
  else
    link m->new_data to d->new_data
  }
```

We now no longer need the `do_transform` variable. Instead we initially create a link for *measure*'s `new_data` events directly to *display*. Whenever *display* sends a `set_transform` event requesting that the transformation be activated, we break the direct link between *measure* and *display*. Now when *measure* generates `new_data` events they will be sent to Glish, which will then transform the data

When Glish executes the first two lines of this script it creates instances of *measure* (running on the host "mon") and *display* (running locally) and assigns to the variables `m` and `d` values corresponding to these Glish clients. Executing the next line:

```
whenever m->new_data do
```

specifies that whenever the client associated with `m` generates a `new_data` event, execute the following statement:

```
    d->new_data( $value )
```

This statement says to send a new event to the client associated with `d`. The event's name will be `new_data` and the event's value is specified by whatever comes inside the parentheses; in this case, the special expression `$value`, indicating the value of the most recently received event (*measure*'s `new_data` event).

The last two lines of the script are analogous; they say that whenever *display* generates a `take_data` event an event with the same name and value should be sent to *measure*.

Our system could easily be a bit more complicated. Suppose that prior to viewing the measurements with *display* we first want to perform some transformation on them. The transformation might for example calibrate the values and scale them into different units, filter out part of the values, or FFT the values to convert them into frequency spectra. Rather than building the transformation into *measure*, we would like our system to be modular, so we use a separate program called *transform*.

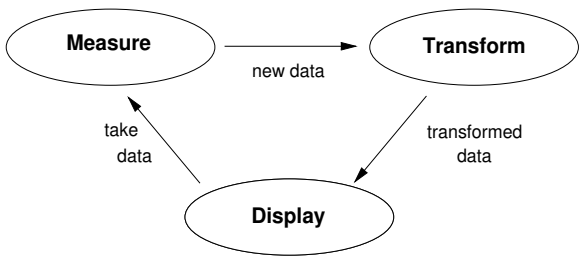

Figure 2: Three-Program Distributed System

Figure 2 shows the flow of control and data in this new system. *measure* sends its values to *transform*; *transform* derives some transformed values and sends them to *display*; and *display* tells *measure* when to take more measurements. With Glish it's easy to accommodate this change:

```
m := client("measure", host="mon")
d := client("display")
t := client("transform")

whenever m->new_data do
    t->new_data( $value )

whenever t->transformed_data do
    d->new_data( $value )

whenever d->take_data do
    m->take_data( $value )
```

The third line runs *transform* on the local host and assigns a corresponding value to the variable `t`. The first `whenever` forwards `new_data` events from *measure* to *transform*; the second `whenever` effectively forwards *transform*'s `transformed_data` events to *display*, but changes the event name to `new_data`, since that's what *display* expects. The third `whenever` is the same as before.

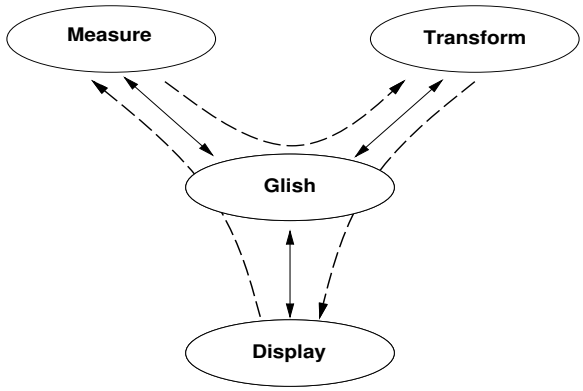

Figure 3: Conceptual Event Flows vs. Actual Flows

An important point in this example is that while *conceptually* control and data flow directly from one program to another, in reality all events pass through the Glish interpreter. Figure 3 illustrates the difference. Here solid lines show the paths by which events actually travel, while dashed lines indicate the conceptual flow. While this centralized architecture doubles the cost of simple "point-to-point" communication, it buys enormous flexibility. For example, suppose sometimes we want to use *transform* before viewing the data and other times we don't. We add to *display* another button that lets us choose between the two. It generates a `set_transform` event with a boolean value. If the value is true then we first pass the measurements through *transform*, otherwise we don't.

To accommodate this change in our Glish program we could add a global variable `do_transform` to control whether or not we use *transform*:

```
m := client("measure", host="mon")
t := client("transform")
d := client("display")
do_transform := T

whenever m->new_data do
    {
    if ( do_transform )
        t->new_data( $value )
```

an "FFT-done" event whose value is two arrays, the Fourier components of the original data. More generally, programs can also spontaneously create events in response to external actions, such as a piece of hardware signalling that some condition has changed, a timer going off, or a person interacting with a graphical interface.

Our software bus, called *Glish*, has three parts:

- a C++ class library that programs (Glish *clients*) link with so they can generate and receive events and manipulate structured data;
- the Glish "sequencing" language analogous to *perl* (but considerably different in flavor);
- an interpreter process for executing Glish scripts and acting as a central "clearinghouse" for forwarding events between processes.

The Glish system is very flexible:

- existing programs can be turned into Glish clients either by writing event-oriented, C++ "wrappers" around them or by encapsulating their filter behavior using `stdin` and `stdout` events;
- clients in a Glish script can run on different computers, which can have heterogeneous architectures;
- Glish provides a full programming language for manipulating the events and data generated by and sent to clients.

In the next section we present an example of the type of systems we want to build with Glish, show how we would use Glish to construct the system, and then present several refinements to convey the flavor of the Glish approach. In Section 3 we give an overview of the more conventional aspects of the Glish language and in the following section discuss those facets of the language concerning event-oriented interprocess communication.

In Section 5 we discuss the C++ class libraries used to integrate programs into the Glish system and give an example of an "FFT" server written using the libraries. The next two sections discuss the implementation and performance of the system. We then conclude with an overview of related work, the present status of the system, and our thoughts on future work.

2 Example of Building a System Using Glish

For an idea of the sorts of problems Glish is meant for and how it's used to solve them, consider a simple example where we want to repeatedly view readings generated by an instrument attached to a remote computer called "mon". Suppose we have a program *measure* that reads values from the special hardware device and converts them into two floating-point arrays, `x` and `y`. *measure* needs to run on the remote host "mon" because that's where the special hardware resides. We have another program, *display*, for plotting the x/y data, which we want to run on our local workstation. *display* also has a "Take Measurements" button that we can click on to instruct the hardware to take a new set of measurements.

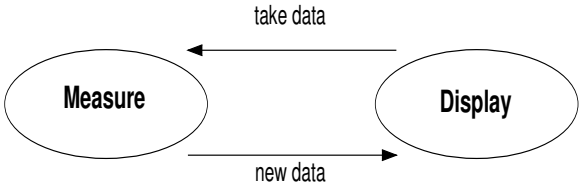

Figure 1: Simple Two-Program Distributed System

The first problem we're interested in is simply to connect together *measure* and *display* so that when *measure* produces new values they're shown by *display*, and when we click the display's button *measure* goes off and reads new values. Figure 1 illustrates the flow of control and data: *display* tells *measure* to take measurements, and *measure* informs display when new measurements are available.

To implement even this simple system under Unix requires constructing a session-layer protocol which then has to be implemented on top of sockets or RPC. When using Glish, though, the protocol and the communication mechanism are built-in. Every program in a Glish system communicates by generating *events*, messages with a name and a value. For our simple system we might write *measure* so that whenever it has new readings available it generates an event called "`new_data`". The value of the event will be a record with two elements, `x` and `y`, the two arrays of numbers it has computed from the raw measurements. We would write *display* so that when it receives a `new_data` event it expects the value of the event to be a record with at least `x` and `y` fields; it then plots those values. Similarly, when we push the "Take Measurements" button *display* will generate a `take_data` event, and whenever *measure* receives a `take_data` event it will get a new set of readings and generate a new `new_data` event.

Here is a Glish script that when executed creates the two processes, one remotely, and conveys their messages to each other:

```
m := client("measure", host="mon")
d := client("display")

whenever m->new_data do
    d->new_data( $value )

whenever d->take_data do
    m->take_data( $value )
```

Glish: A User-Level Software Bus for Loosely-Coupled Distributed Systems

Vern Paxson* and Chris Saltmarsh†
Lawrence Berkeley Laboratory
One Cyclotron Road
Berkeley, CA 94720

Abstract

We describe *Glish*, an interpreted language for building distributed systems from modular, event-oriented programs. These programs are written in conventional languages such as C, C++, or FORTRAN. Glish scripts can create local and remote processes and control their communication. Glish also provides a full, array-oriented programming language for manipulating binary data sent between the processes. In general Glish uses a centralized communication model where interprocess communication passes through the Glish interpreter, allowing dynamic modification and rerouting of data values, but Glish also supports point-to-point links between processes when necessary for high performance. Glish is available via anonymous ftp.

1 Introduction

Much of the power of Unix stems from the ways in which users can combine different programs. The notions of standard input and output, pipelines, filter programs, and command shells all encourage the creation and use of modular programs that can be "plugged together" in novel ways. Traditionally Unix command shells have focussed on creating and connecting together processes. Recently, however, command shells such as *perl* [WS91] also provide powerful languages for manipulating the output generated by programs. Often a *perl* user can write a considerable portion of a task in *perl*, rather than needing to create new filter programs. We might say that in this regard *perl* provides better "glue" than previous shells for connecting together programs.

There are some limits, however, to the power of Unix pipelines, even when augmented with a shell like *perl*. Data in pipelines flows in only one direction; two programs cannot communicate with each other back and forth. Furthermore, the data the programs manipulate is generally limited to character streams whose structure is column- or line-oriented. Communicating large quantities of numeric data is inefficient at best and inaccurate unless care is taken. Communicating *structured* data—collections of related values, perhaps with different types—is particularly difficult.

While it is possible to circumvent these restrictions, the only support for doing so is at the operating-system-call and run-time-library level. There is no analog of shell programming for interconnecting processes so they can communicate in complex ways and share binary, typed data.

Applications such as simulation systems often can be well modeled as a number of separate processes perhaps running on different hosts that occasionally send structured data back and forth; i.e., as loosely-coupled distributed systems. Since the present facilities in Unix provide little high-level support for such an approach, one instead often resorts to writing the system as a set of processes that have considerable knowledge about what other processes and data structures exist in the system. This system-specific knowledge makes it difficult to extend the system in unforeseen ways, so unless one has a complete understanding of the system requirements at the outset, one is likely to find the final system uncomfortably restrictive.

In this paper we discuss a *software bus*-style solution to building flexible, loosely-coupled distributed systems. The main thrust of the software bus approach is that individual programs should be wholly modular, with no knowledge of other programs or data types that might exist in the system. The software bus supplies a uniform way for programs to communicate without knowing about one another. In our system, programs are written in terms of *events*, which are name/value pairs. In the usual case, programs receive an event, perform some sort of action in response to the event, and possibly generate one or more new events associated with the response. Such programs are similar to RPC servers, except that "calls" to the programs are not synchronous. An example is an FFT server, which might be sent an event with the name "please-FFT-this" and an associated value of an array of double precision data, to which the server in turn generates

*Work supported by the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

†Superconducting Super Collider Laboratory, Operated by the Universities Research Association, Inc., for the U.S. Department of Energy under Contract No. DE-AC02-89ER40486.