SWILL: A Simple Embedded Web Server Library

Sotiria Lampoudi and David M. Beazley
Department of Computer Science
University of Chicago
Chicago, Illinois 60637
{slampoud,beazley}@cs.uchicago.edu

Abstract

We present SWILL, a lightweight programming library that adds a simple embedded web server capability to C and C++ programs. Using SWILL, it is possible to add Internet accessibility to programs that are poorly matched to more traditional methods of web programming such as CGI scripting or web server plugin modules. SWILL also makes it easy for programmers to add web-based monitoring, diagnostics, and debugging capabilities to software not normally associated with internet programming. We like to think of SWILL as an attempt to turn the problem on its head: traditionally, the web server came first, the "programs" later; in our approach, the application is written first, and the server integrated last. For some types of applications, this approach is far more painless. In this paper, we provide an overview of the SWILL library and describe how we have used it to provide web access to a variety of applications including scientific simulation software, a compiler, and a hardware emulator for teaching operating systems.

1 Introduction

With the growth and popularity of the web, many software developers are interested in building web-based interfaces to their applications. However, much of the effort involved in doing this seems to be spent figuring out how to integrate an application into an existing web server using CGI scripts, server plugin modules, or some sort of middleware layer. Although it is certainly possible to implement a web interface in this manner, a programmer may have to contort their application or implement a complicated runtime environment to make it work. To further complicate matters, there are certain applications in which web access would be useful, but which do not really fit into the standard mold of an internet application. Examples might include simulators, long running scientific programs, and compilers.

Instead of figuring out how to integrate an application with a web server, an alternative approach might be to flip the whole situation around and to ask what it would take to add a web server to an application as a programming library. If a web server could be added as a library, it might greatly simplify the task of creating web-accessible programs. For instance, since the the web server would be part of the application itself, there would be no need to worry about the installation and configuration of a complicated web server environment. A library would also make it easier to add web capabilities to programs that are not really intended to be internet applications, but where a web interface would be useful in providing remote accessibility or as a simple diagnostics tool.

In this paper, we describe SWILL (Simple Web Interface Link Library). SWILL is a lightweight library we have developed that makes it easy to add a web server to C/C++ programs. It consists of a handful of C functions that are inserted into an application to make it web accessible. The implementation is intentionally minimal and is primarily intended for developers who would like to create a web interface without a lot of hassle.

SWILL was initially written for use with high performance scientific simulation software [4]. However, the implementation is generic enough to be used in a variety of other applications–limited only by the programmer's imagination. For example, we have used SWILL in a compiler project to help with the debugging of parse tree data. It has also been used to monitor the internals of a hardware simulator for teaching operating systems.

The rest of this paper provides a brief overview of SWILL followed by some examples of how we have used the library. At the end, we provide a brief overview of related work and future plans.

2 Library Introduction

The best way to introduce SWILL is with a simple example. Although the library contains about 25 functions, only a handful are needed to get started as shown in this “Hello World” example:

```
#include <swill.h>
void hello(FILE *f) {
   fprintf(f,"Hello World!\n");
}

int main() {
   swill_init(8080);
   swill_handle("hello.txt",hello,0);
   while (1) {
      swill_serve();
   }
}
```

In this example, the web server is activated by calling `swill_init()` with a TCP port number. One or more documents are then registered with the server using `swill_handle()`. `swill_serve()` is then used to wait for a connection. When a connection is received, an appropriate handler function is invoked depending on the URL. In this case, a request for the document “`hello.txt`” invokes the `hello()` function and produces the output that you expect.

If an application wants to perform other work in addition to checking for connections, a non-blocking polling function is used. For example:

```
while (1) {
   swill_poll();  /* Requests? */
   ...
   /* other work */
   ...
}
```

This approach allows the web interface to be easily inserted into applications that provide their own event or computation loops. For example, a scientific simulation might call `swill_poll()` at selected intervals during computation, but spend the rest of its time crunching numbers.

There are no restrictions on the type of output a SWILL handler function may produce. However, the document type is implicitly determined by the suffix supplied to the `swill_handle()` function. For example, if a function produces HTML, the following code would be used:

```
void hello(FILE *f) {
   fprintf(f,"<HTML><BODY>\n");
   fprintf(f,"<b>Hello World!</b>\n");
   fprintf(f,"</BODY></HTML>\n");
}

int main() {
   ...
   swill_handle("hello.html",hello,0);
   ...
}
```

Similarly, the following code produces a PNG image using the freely available GD library [7]:

```
void
image(FILE *f) {
  int black,white;
  gdImagePtr im;

  im = gdImageCreate(64,64);
  black = gdImageColorAllocate(im,
                  0,0,0);
  white = gdImageColorAllocate(im,
                  255,255,255);
  gdImageLine(im,0,0,63,63,wht);
  ...
  gdImagePng(im,f);
  gdImageDestroy(im);
}

int main() {
   ...
   swill_handle("image.png",image,0);
   ...
```

Since a programmer may want to reuse the same handler function for different web pages, SWILL allows an optional pointer to be passed to the handler functions. This pointer is used to pass application specific data or an object to the handler. For example, an application that created various types of data plots might look roughly like this:

```
void
make_plot(FILE *f, void *clientdata) {
  Plot *p = (Plot *) clientdata;
  ...
  // Generate plot
  ...
  write_plot(p,f);
}

int main() {
  ...
  e = new energy_plot();
```

```
  d = new density_plot();
  ...
  swill_handle("energy.png",make_plot,e);
  swill_handle("density.png",make_plot,d);
   ...
```

Although SWILL is primarily intended for dynamic content generation, it can also deliver individual files or files from a user-specified directory. For example, if a programmer wanted to register a specific file with the server, they would do this:

```
swill_file("foo.html","./htdocs/foo.html");
```

Similarly, a directory of files is registered as follows:

```
swill_directory("./htdocs")
```

When a directory is registered, SWILL delivers files much like a traditional web-server.

In certain applications, a programmer might want to receive HTTP query variables as input parameters (as might be supplied from an HTML form). SWILL automatically parses HTTP query strings in both GET and POST requests. To access query variables as strings, the following function is used:

```
char *swill_getvar(const char *name);
```

However, a more convenient way to get form variables is to use `swill_getargs()` as shown in this example:

```
void adder(FILE *f) {
   double x,y;
   if (!swill_getargs("d(x)d(y)",&x,&y)) {
     fprintf(f,"Missing values!\n");
     return;
   }
   fprintf(f,"%g + %g = %g\n", x,y,x+y);
}
```

The argument to `swill_getargs()` is a format string that specifies the types and names of form variables to be converted. If available, the variables are decoded, placed into C variables, and a success code returned.

3 Concurrency and I/O

SWILL is a single-threaded server that does not rely upon concurrency mechanisms such as forking or multithreading. This limitation is by design and is related to the goal of having a server that could be easily embedded into a variety of specialized applications such as parallel scientific codes, hardware emulators, and so forth. In these situations, the use of concurrency can introduce serious reliability and portability problems. For instance, an application may not be thread-safe, making it impossible to reliably execute a handler function in parallel with normal execution. Similarly, forking may be impractical in applications with heavy resource utilization or which rely upon interprocess communication (e.g., message passing).

Because of the single-threaded execution model, the implementation of SWILL relies entirely upon non-blocking I/O with timeouts. This prevents the server from indefinitely blocking the application in the event of bad connections and missing data. For instance, if a connection is made, but no HTTP headers are received, SWILL automatically closes the connection after a timeout and returns. The timeout is fully configurable by the user and can be set to only a few seconds if desired.

The I/O for handler functions relies upon a temporary file created with `tmpfile()`. When requests are serviced by handler functions, output is placed into this file. When a handler function has finished execution, the file contents along with HTTP headers are sent back to the client. Normally, SWILL simply passes a corresponding `FILE *` object to handler functions so that they can perform I/O. As an optional feature, SWILL can also capture standard output. This is enabled in `swill_handle()` by prefixing the document name with `stdout:` as follows:

```
swill_handle("stdout:foo.html",foo,0);
```

In this case, all I/O operations on `stdout` are captured for the duration of the handler function. This capture is sufficiently powerful to allow other programs to be executed using `system()` and to have the output of those programs redicted to a web page. For example, the following handler function would capture the output of a system command:

```
void listfiles() {
    system("ls -l");
}
...
swill_handle("stdout:files.txt",listfiles,0);
```

4 Security and Reliability

SWILL is not appropriate for use in applications that require a high degree of security since no support for SSL or cryptographically secure user authentication is provided. However, the library does provide a few simple security features to restrict access. First, basic HTTP user authentication is available by registering names and passwords like this:

```
swill_user("dave","iluvschlitz")
```

Second, IP filters can be used to disallow or allow connections from specific IPs or ranges of IPs. For example:

```
swill_allow("127.0.0.0");
swill_deny("128.135.11.");
swill_deny("128.135.11.8");
swill_deny("");
```

SWILL also allows users to register a log file in which requests will be recorded and which can be monitored to check for suspicious activity.

Due to the lack of concurrency, a SWILL application may be vulnerable to a denial of service attack. However the library does take reasonable precautions to allow an application to make progress. As mentioned in the previous section, all I/O operations involve non-blocking system calls with timeouts. Therefore, it is not possible for a client to indefinitely block execution by keeping the connection open without transmitting any data. SWILL is also quick to close connections if it detects malformed data such as bad HTTP headers or garbled input. We have considered the possibility of automatically blocking IP addresses that repeatedly send bad requests. However, this is not implemented at this time. Given that IP filters can be used to block access, a user already has the means to restrict access to a set of known hosts.

Finally, since SWILL is embedded in an application, it is certainly possible for bad programming to break the server. For instance, a poorly written handler function could enter an infinite loop or start a computation that exceeds available machine resources. In this case, the application would become unresponsive and would probably die. SWILL does not take any steps to prevent such problems. However, these can be anticipated with a certain amount of common sense, error checking, and having an understanding of the underlying execution model of the application.

5 Parallelized SWILL

A very useful feature of SWILL is that it supports SPMD-style parallel applications that utilize MPI (MPICH is currently supported) [8]. This allows it to be used on Beowulf clusters and large parallel machines. If used in this style, every node calls `swill_poll()` in parallel which results in a global synchronization. If an incoming request is received, it is forwarded to all of the nodes which then execute the handler function in parallel.

Using SWILL in this setting is no more difficult than in the single processor setting; the HTTP client connects to the master node of the computation, issues a request and receives sorted output collected from all nodes. Under the hood the implementation is also quite simple. The master node (`MPI_Rank == 0`) receives requests and broadcasts them to all of the other nodes. Each node runs the handler function in parallel after which the content is collected by the master node and served in a coherent manner to the HTTP client.

In parallel scientific programming performance is the foremost consideration. To evaluate the performance of `swill_poll()` we distinguish three cases: a) the case in which there is no pending HTTP request, b) the case where there is a pending request for a file or authentication, and c) the case where there is a request for dynamic content.

a) This case is quite simple. If there is no pending HTTP request, the master node communicates a null request to all nodes, and the host code continues execution immediately after the call to `swill_poll()`. There is one synchronization in this case, for the call to broadcast the null request.

b) This case is also handled with an overhead of one synchronization. The master node identifies the pending request as one that can be served by it alone, serves it and broadcasts a null request. This also reveals one of our underlying assumptions, namely that the SPMD program is running on a shared filesystem of some sort, so that it would make no sense to return n (for n nodes) copies of the requested file. This behavior is easy to get around. If it is desirable that a copy of the requested file be returned from *each* node, the user can just write a function that reads the file in and prints it to stdout or uses `swill_printf()`.

c) This is a somewhat more expensive operation. The master node broadcasts the pending request to all nodes, who parse it and execute the appropriate function. Then each node transmits the result of its execution to the master. Next, the master node serves the HTTP response and resumes execution of the host code, while the back-end nodes resume computation immediately. All in all, this case has a cost of one broadcast plus n (for n nodes) communications.

In our quest for simplicity we have left it up to the user to produce output that denotes what process each segment of the output is coming from – SWILL merely orders it in rank order and serves it.

One interesting use of the parallel capability of SWILL is in overcoming limitations – whether due to architecture or policy – imposed by the administrators of clusters and supercomputing centers. Often it is not possible to access the backend computational nodes of a supercomputer or cluster in order to query the state of one's execution – at least not through a normal shell. When an application embeds a web server, all that is required is knowledge of the master node and access to an unprivileged port. Web requests can then be easily translated into operations that execute on each node of the system.

6 Applications

SWILL has primarily been developed as a tool for remote process monitoring, debugging, and diagnostics. This section describes some applications in which we are using the library.

6.1 Scientific simulation monitoring

The motivating application for most of SWILL's development has been that of monitoring long-running scientific simulations. These programs are typically non-interactive batch jobs that provide little in the way of user feedback. However, to make sure a program is running correctly, a scientist may want to periodically monitor the state of their programs. For example, they might want to check for numerical instabilities or to see how far an experiment has progressed.

To do this, a simulation can be modified slightly by implementing a few handler functions and inserting `swill_poll()` calls into selected places in the simulation loop. For example:

```
/* Initialize SWILL */
swill_init(3737);
/* Register a few handlers */
swill_handle(...);
swill_handle(...);
...
for (i = 0; i < nsteps; i++) {
    compute_forces();
    integrate();
    boundary_conditions();
    redistribute_data();
    if (!(i % output_freq)) {
        write_output();
    }
    /* Check for connections */
    swill_poll();
}
```

Although this is only a simple example, this technique is easily extended to provide a variety of advanced monitoring capabilities. For instance, if a graphics library is available, the web interface can be used to generate on-the-fly plotting and data visualization. Web access can also be provided to temporary files and other debugging output as the simulation runs. Using HTTP query variables and forms, a scientist could even alter various simulation parameters, enable diagnostic features, temporarily suspend computation, and so forth.

6.2 Compiler parse tree browsing

As a more unusual example, we have used SWILL to provide a web interface to SWIG, a compiler for creating scripting language extensions [3]. One of the challenges of compiler implementation is that of creating, traversing, and managing parse tree data. For the purposes of debugging, it is fairly common to dump the parse tree into a text file where it can be examined. Unfortunately, even for small input files, this might generate a lot of output since a parse tree might contain hundreds to thousands of nodes. This makes it difficult to find the specific information of interest.

As an alternative to dumping the parse tree to a file, a more convenient way to examine the parse tree data is to run SWIG using a special `-browse` option like this:

```
$ swig -browse -c++ -python example.i
SWIG: Tree browser listening on port 4908
```

In this mode, the compiler enters a web-server mode after all parsing and code generation stages have been completed. Then, by pointing a browser at the appropriate port number, it is possible to point-and-click through internal parse tree data. A sample screenshot of this interface is shown in Figure 1. Unlike the information in a text dump, the web interface provides a more more detailed picture of how

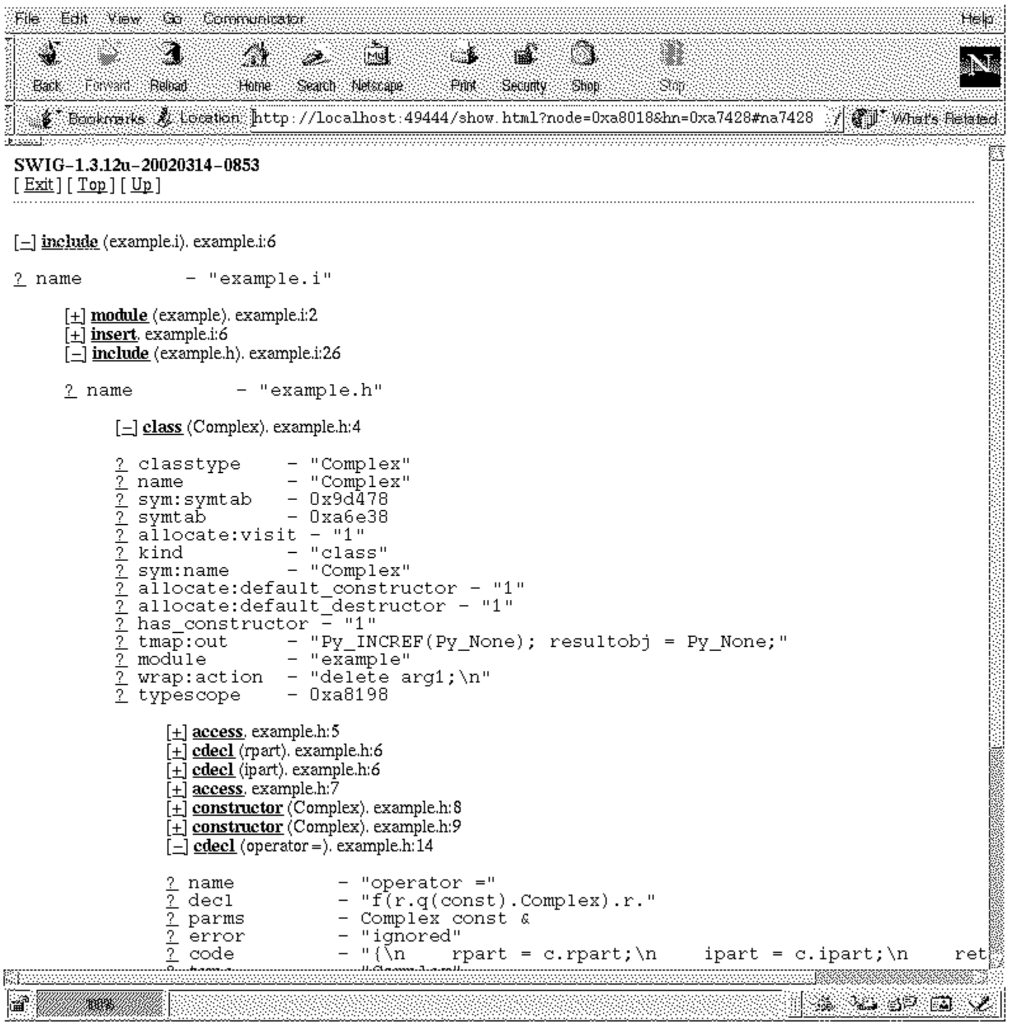

Figure 1: Parse-tree browsing in SWIG

information is organized in the compiler. For instances, pointers are represented by hyperlinks and clicking on a link is essentially the same as dereferencing a pointer in C. The web interface also provides access to compiler symbol tables, type tables, and other information.

At first glance, the idea of a web-enabled compiler sounds crazy. However, this capability greatly simplifies debugging and development of the compiler. Furthermore, a web browser interface works perfectly well as a simple data exploration tool and it was extremely easy to implement—requiring only a half day of effort and a few hundred lines of code. In comparison, the development of a customized tree browser using a GUI toolkit such as Tcl/Tk would have been a much more involved project, would have greatly complicated the configuration of the compiler, and would have provided little if any extra functionality.

6.3 Operating systems project

At the University of Chicago, the course in operating systems requires students to implement a simple Unix-based kernel that runs within an emulated hardware environment. For emulation, we use a modified version of Yalnix, an emulator originally developed by Dave Johnson at Rice University [11]. In this project, students are given eight weeks to implement a kernel from scratch including boot loading, virtual memory management, I/O device drivers, processes, and interprocess communication. Kernels are implemented in C and typically consist of a few thousand lines of code.

One of the problems of working with the emulator is that debugging is extremely difficult. For the most part, the only available diagnostics are an optional hardware trace file and the output of print statements included in the student's implementation. Unfortunately, this information is often incomplete–making it very difficult to reconstruct the system state that might be the source of a problem.

As an experiment, we have recently used SWILL to add a web-interface to the Yalnix emulator. Internally, Yalnix relies heavily on advanced features of signal handling on Solaris. Specifically, user-mode programs execute natively on the SPARC whereas the student kernel runs in response to signals such as `SIGSEGV` and `SIGALRM` (which are transformed into "hardware interrupts"). To instrument the emulator with a web server, we implemented a number of handler functions, initialized the server on startup, and placed a `swill_poll()` call into the `SIGALRM` handler that is used for internal timing of the emulator.

Using the web interface, it is possible to directly connect to the emulated hardware. Available information includes the current status of all hardware registers, page table settings, and the contents of physical memory pages. It is also possible to pause execution and to obtain traces of recent hardware operations (the history of memory mapped I/O ports, registers, and interrupts). More importantly, the web interface allows you to observe system behavior that is nearly impossible to obtain otherwise. For instance, by watching page tables you can easily spot kernel memory leaks and other inefficiencies in the implementation.

7 Discussion of Related Work

Internet programming is obviously a huge topic, making a detailed comparison of SWILL to related work difficult. Overall, we feel that SWILL differs from other work in a number of respects. First, a considerable amount of attention has been given to programming techniques such as CGI scripting, web server modules, server pages, and programming environments for building Internet applications [10, 2, 6, 12, 17]. Although these techniques are successfully used to incorporate programming libraries and other applications into a larger Internet framework, SWILL has a somewhat different focus than this. Instead of trying to build Internet applications, SWILL is mostly concerned with providing Internet access. This may be a subtle point, but the applications for which SWILL is the most useful are not really designed for the Internet—Internet access is merely an add-on feature that can enhance them.

SWILL might also be compared to work in distributed computing. For instance, SOAP and XML-RPC are often mentioned as mechanisms for adding internet access to applications [13, 16]. In fact, toolkits such as gSOAP can be used to simplify the integration of an application into such an environment [14]. The problem with these approaches is that they are mostly focused on the problem of turning an application into some sort of pluggable network service to be used within a complicated middleware layer. SWILL, on the other hand, is much more lightweight and is oriented more towards end-users. For instance, users merely connect to the server and are presented with application-specific information in a format that is easy to use and manipulate. There is no hidden application framework or network layer at work.

Finally, SWILL is closely related to domain-specific efforts in providing remote access to applications. For instance, in scientific computing, a lot of attention has been given to the area of "computational steering" [15]. One of the primary goals of steering research is to provide fine-grained interactivity and user feedback to scientific software that is normally batched-oriented and non-interactive. Traditionally, this work has relied upon customized network protocols, complicated client software, and high-end hardware such as graphics workstations. (For a detailed discussion of computational steering we refer the reader to the excellent article [9].)

In some cases, application frameworks may provide web access for this purpose. The Cactus code [5, 1], a modular scientific programming framework, is such an example (the CactusConnect/HTTP and CactusConnect/HTTPExtra "thorns" – modules – are supposed to provide HTTP access to a cactus application). In such cases, however, the web access features are not usable as a stand-alone library. If one wants to have web access, one is forced to buy into a framework with all the pain and risks that entails. SWILL tries to avoid this by focusing exclusively on the problem of web access.

8 Implementation Details

SWILL is implemented entirely in ANSI C and consists of about 2500 semicolons. Most of the implementation (1500 semicolons) simply provides a small set of generic data structures (hashes, lists, strings, etc.) that are used elsewhere. The library itself requires minimal memory overhead. However, all of the generated web pages involve internal buffering. Therefore, memory use is directly proportional to the size of generated web pages. Clearly the library would be inappropriate for serving huge amounts of data. However, this was not a design goal.

The performance overhead of using SWILL depends on frequency of polling (and obviously the number of incoming connections). On a single CPU, `swill_poll()` is nothing more than a thin wrapper around the `select()` system call. With MPI, polling requires a barrier synchronization across processors. This is obviously more expensive and careful consideration must be given to parallel applications.

For networking, SWILL relies upon the HTTP/1.0 protocol. Although this is less powerful than HTTP/1.1, it is easier to implement and perfectly well suited for most situations.

9 Limitations

SWILL was primarily designed as a tool for building quick-and-easy web interfaces to C/C++ programs. Its single threaded execution would be inappropriate for a high-traffic web site and you probably would not want to use it as the basis of a large internet application. Similarly, internal buffering and other aspects of the implementation make the server inappropriate for delivering very large amounts of data. The server is also unable to support data streaming or any sort of application in which the HTTP connection would be kept alive over a prolonged period.

Although SWILL provides some basic security mechanisms, it would not be appropriate for applications in which security was critical. It should also be added that firewalls and other security mechanisms may prevent users from accessing a server if access to user TCP ports is blocked. Obviously, SWILL cannot address these problems of social engineering.

10 Future Plans

In our own work, SWILL has proven to be remarkably simple and effective to use. Therefore, we have every intention of preserving the minimal nature of the implementation. However, it may be interesting to provide alternative interfaces to C++ and Fortran. Since SWILL does not rely upon anything more than a few simple functions and standard I/O operations, it would be relatively easy to implement handler functions with a slightly different calling convention. For example, in C++, SWILL might encapsulate I/O in an iostream object instead of a `FILE *`.

We have also considered the idea of allowing each SWILL-server to "phone home" to a user-defined master server. If a user was running many different web-enabled applications, this scheme might make it easier to keep track of where they are running. For example, a user could simply connect to the master server and jump to a specific application from there.

Obvious improvements could be made to the underlying HTTP protocol such as support for HTTP/1.1, secure sockets, and digest-based user authentication. However, supporting such features would introduce a lot of extra complexity and would probably offer only marginal benefits in return.

11 Availability

SWILL is freely available under a LGPL licence. More information is available at:

`http://systems.cs.uchicago.edu/swill`

12 Acknowledgments

We would like to thank the reviewers for their helpful comments. Mike Sliczniak and Hasan Baran Kovuk contributed to early parts of the SWILL implementation.

References

[1] G. Allen et. al., *Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus*, Supercomputing, Denver, CO, (2001).

[2] Apache Web Server, `http://www.apache.org`.

[3] D.M. Beazley, *SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++*, 4th Annual Tcl/Tk Workshop, Monterey, CA (1996).

[4] D.M. Beazley and P.S. Lomdahl, *Controlling the Data Glut in Large-Scale Molecular Dynamics Simulations*, Computers in Physics, Vol. 11, No. 3. (1997), p. 230-238.

[5] Cactus Code, `http://www.cactuscode.org`.

[6] C Server Pages, `http://tesitra.com/cserverpages`.

[7] GD, `http://www.boutell.com/gd/`.

[8] W. Gropp, E. Lusk, A Skellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, (1999).

[9] W. Gu, J. Vetter, K. Schwan. *Computational steering annotated bibliography*, Sigplan notices, 32 (6): 40-4, (1997).

[10] S. Gundavaram, *CGI Programming*, O'Reilly & Associates Inc., (1996).

[11] D. B. Johnson, *Yalnix: An Undergraduate Operating System Course Environment and Project Set*, Department of Computer Science, Rice University.

[12] E. Meijer and D. van Velzen,*Haskell Server Pages: Functional Programming and the Battle for the Middle Tier*, Proc. Haskell Workshop, (2000).

[13] SOAP, `http://www.w3.org/TR/SOAP/`.

[14] R.A. van Engelen, K.A. Gallivan, *The gSOAP Toolkit for Web Services and Peer-to-Peer Networks*, Proc. IEEE CC Grid Conf., (2002).

[15] J. Vetter, K. Schwan, *High Performance Computational Steering of Physical Simulations*, Proc. Int'l Parallel Processing Symp., Geneva, pp. 128-132, (1997).

[16] XML-RPC, `http://www.xmlrpc.com`.

[17] Zope, `http://www.zope.org`.