An Easily Embeddable Web Server for Remote Simulation Monitoring

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

Abstract

We present an embedded web server library known as SWILL (Simple Web Interface Link Library) that facilitates the addition of a remote monitoring capability to scientific applications by the scientists themselves. Unlike framework-based solutions, SWILL utilizes a collection of low-level I/O filter functions that make it possible to reroute nearly arbitrary program output to a collection of dynamically generated web pages. This approach not only simplifies the use of the library, but makes it possible to add an embedded web server to existing programs with very few modifications to the source code. In addition, the library fully supports SPMD-style MPI programs and allows for the parallel generation of web pages. We provide an overview of SWILL and describe how it can be incorporated into existing scientific software.

1 Introduction

For the past decade, considerable attention has been given to the problem of building more interactive, extensible, and user-friendly scientific software. For example, efforts in computational steering, distributed computing, and scripting languages all claim to break the traditional batch processing cycle and provide software with greater flexibility [11, 8, 5, 3]. However, despite the success of these efforts, the simple fact remains: large-scale production simulations run uninterrupted for tens to hundreds of hours on the largest machines available.

A common problem faced by scientists conducting these long-running experiments is that of monitoring their progress long after the interactive interface has been detached or the program has been submitted as a batch processing job. For example, a scientist may want to examine the simulation state to see if a numerical instability has occurred (and to stop the job if necessary). Similarly, it is often useful to obtain diagnostic information or visualizations of live simulation data at intermediate stages of the computation. In some cases, it may be useful to make carefully controlled adjustments to the running computation in some manner; for example, a scientist might want to change the frequency at which output files are generated or to start tracing a parameter of interest. Unfortunately, most simulation codes do not allow this sort of user interaction after a simulation has started. In fact, in many cases the only way to monitor a simulation is to examine its log files or directories for the presence of new output. Even then, it can be difficult to obtain an accurate picture of simulation state unless files are offloaded to another machine and examined with a data analysis tool.

A very simple and practical solution to the monitoring and control problem is to instrument simulation software in a way that allows users to access the running simulation through an ordinary web browser. For example, a simple web-based simulation monitor using scripting language techniques is briefly described in [2]. Similarly, many advanced simulation frameworks such as Cactus now provide special web server modules [6]. Unfortunately, the main problem with these approaches is that their web interfaces are often tightly coupled to the underlying framework. As a result, it may be difficult to apply this work to existing scientific software–much of which may be home grown, adapted from so-called "legacy" systems, or custom tailored to environments not supported by the particular framework.

To address this limitation, we have developed an easy to use library called SWILL (Simple Web Interface Link Library), that is designed to add an embedded web server capability to existing scientific software. Unlike framework-based solutions, a

key feature of the implementation is its minimalistic API in which the output of standard I/O library functions is intercepted and rerouted to dynamically generated web pages. Hence, simulation functions need not be altered to include SWILL-specific I/O calls, as we trap their output and reroute it to the web browser. In addition, the library fully supports large scale message passing applications written using MPI.

2 Scientific Software

Consider the implementation of a typical scientific simulation: First, there are procedures for numerical integration, matrix solvers, boundary conditions, initial conditions, and data management. Next, there is a time-stepping loop that actually runs the steps of the simulation. For example:

```
for (i = 0; i < nsteps; i++) {
    compute_forces();
    integrate();
    boundary_conditions();
    redistribute_data();

    if (!(i % output_freq)) {
       write_output();
    }
}
```

Within the time-stepping loop, there are calls to I/O functions responsible for producing checkpoints, data files, images, and other types of output. Finally, an application often defines a variety of debugging and diagnostic functions that are used to help verify the correctness of the application. These functions do not typically appear in the inner simulation loop. However, they may be enabled during application development or while debugging simulation parameters. For example, in a molecular dynamics code, a debugging function might be written to look at the distribution of atoms across subcells so that a measure of load-balancing could be obtained:

```
void debug_cells() {
  /* Examine cell structure */
  ...
  printf("Max per cell : %d\n",max);
  printf("Avg per cell : %g\n",avg);
  printf("Empty cells  : %d\n",nempty);
   ...
}
```

A critical aspect of the implementation is that even though execution is tightly controlled by the time-stepping loop, a considerable amount of additional functionality is contained within the application. Furthermore, much of this functionality represents the types of operations a scientist would most want to access through a web interface. For instance, even though it makes no sense to repeatedly call a debugging function as part of the inner loop, it may be extremely useful to periodically call such a function from a web interface. Similarly, it may be useful to utilize some of the existing I/O procedures to produce on-demand snapshots of simulation data.

3 Web Servers

In order for a simulation to function as a web server, it must create a network socket and listen for incoming HTTP requests. The form of a typical request is shown in Figure 1. Once received, the request is parsed into a URL, a collection of HTTP headers, and a set of form variables (if any). The URL is always contained in the first line of the request and may optionally include URL-encoded form variable values as shown. The HTTP headers follow the first line and are terminated by a blank line. Although most of the headers can be ignored, they are sometimes used to control other aspects of the connection such as caching and authentication. For instance, in the figure, the "Authorization:" field contains a base-64 encoded *name:password* string that a server could decode to verify the identity of a user.

To send a response back to the browser, the server writes a set of HTTP headers followed by the raw data back to the client as shown in Figure 2. A minimal response includes information about the protocol, file type, and content length.

Although the implementation of a simple web server is a straightforward exercise, it adds considerable complexity to the traditional file I/O model found in most simulations. A user implementing a web server from scratch, has to worry about the details of network programming, parsing of requests, decoding of special data formats, user authentication, and issues related to error handling. In addition, responses must be annotated with HTTP metadata such as MIME types and length fields (which for dynamically generated output, can not be determined until after all of the output has been created). On a parallel machine, additional implementation difficulties arise. For instance, it probably does not make sense to run an independent web

```
GET /foo.html?step=1000&file=dat%2E45 HTTP/1.0
Referer: http://schlitz:8080/info
Connection: Keep-Alive
User-Agent: Mozilla/4.73 [en] (X11; U; SunOS 5.8 sun4u)
Pragma: no-cache
Host: schlitz:8080
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Authorization: Basic ZGF2ZTpmb28xMjM=
```

Figure 1: Example of an HTTP request

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 17283
Connection: close
Server: SWILL/0.1

<HTML>
<HEAD>
...
</HTML>
```

Figure 2: Example of an HTTP response

server on each node of the machine due to the difficulty of coordinating its operation with the parallel execution model of the entire application. Therefore, an application might run a server on a master node (node 0) and have it coordinate actions with all of the other nodes using MPI primitives. In this case, the master node serves as a proxy responsible for broadcasting the incoming request to the rest of the nodes and collecting their output in a way that allows a reasonable response to be produced.

Although a web server interface can be implemented in a library, there is a tendency for such libraries to provide their own set of dedicated I/O functions. This forces significant portions of the application to be modified and results in the development of additional code that largely mirrors the functionality of existing I/O procedures. Clearly this complicates software development and limits the usefulness of the web interface since scientists are unlikely to support two different I/O models during the early stages of application development (which is precisely when a remote monitoring capability is most useful).

4 SWILL Overview

SWILL takes an alternative approach to the scientific web server problem. Rather than providing a large programming API or a web-enabled problem solving framework, it overrides the default behavior of common I/O operations and allows already existing functions to generate web pages. For example, to adapt our earlier example to SWILL, a scientist could modify the simulation loop as follows:

```
swill_init(3737);
swill_handle("stdout>cells.txt",
             debug_cells,0);

for (i = 0; i < nsteps; i++) {
    compute_forces();
    integrate();
    boundary_conditions();
    redistribute_data();

    if (!(i % output_freq)) {
       write_output();
    }
```

```
    /* Check for connections */
    swill_poll();
}
```

In this example, the `swill_init()` function opens a network port for receiving incoming connections. The `swill_handle()` function registers a document "cells.txt" and specifies that the function `debug_cells()` should be called when a request for this document arrives. The special modifier `>stdout` specifies that anything written to standard output should be redirected to the web-page during the execution of the handler function. Finally, the function `swill_poll()`, inserted into the simulation loop, checks for incoming connections and serves requests (if any).

With these minor modifications, a scientist will find that their application has now been turned into a simple web server. Furthermore, when they request the document "cells.txt", they will receive the output of `debug_cells()` even though no modifications were made to that function. Moreover, the scientist will find that none of the default I/O behavior of their application has been changed. For instance, if the `debug_cells()` function is called from outside the web server, its output is directed to standard output in the normal fashion.

SWILL also provides support for more standard types of web server operations. For example, a user might add the following code to serve an individual file, to serve a directory of files, or to add user authentication:

```
swill_init(3737);
swill_file("desc.html","./html/desc.html");
swill_directory("./htdocs");
swill_auth("beazley","pfm123");
```

Thus, using these functions, the web interface to an application can be built using a collection of static HTML files and a set of handler functions responsible for creating dynamic output.

Although SWILL is easy to use with functions that produce text, it is also possible to produce images using common graphics libraries. As an example, consider the use of gd; a simple open source graphics library for producing PNG and JPEG images [4]. To create a simple PNG image file, a scientist might write a general purpose function like this:

```
void
make_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,white);
  ...
  gdImagePng(im,f);
  gdImageDestroy(im);
}
```

This function could be used as a normal part of the simulation loop (e.g., to produce images at periodic intervals). However, the function can also be used in the web interface by registering it with SWILL as follows:

```
swill_handle("image.png",make_image,0);
```

In this case, access to the document "image.png" invokes the `make_image` function which create a PNG image. This image is then be captured by SWILL and returned as a web page–all without creating any temporary files or having to modify parts of the gd library.

5 Execution Model

SWILL is based entirely on a polling model of execution. In order to use the library, an application must first initialize the library and then make explicit calls to `swill_poll()` at selected points of execution. Although rarely used in conventional client-server applications, polling is particularly well suited for scientific applications. For one, a scientist would rarely want a remote server to return results based on inconsistent or partially computed simulation data (e.g., a snapshot taken in the middle of numerical integration step). Nor would a scientist want the web server interface to consume so much CPU time that it negatively affects application performance. Thus, polling allows a scientist to precisely control the points at which it is safe to examine simulation data as well as the frequency at which the web interface will be monitored for incoming connections. Another reason to use polling is that on parallel machines, the web server interface may want to invoke functions that utilize MPI functions in their underlying implementation. Given that MPI libraries are generally not thread-safe, nor can MPI functions be used within a collection of child processes created with fork, it is not practical to construct a web server library that supports concurrent operation in this environment.

When requests are received by the server library, they are parsed into a URL, a collection of HTTP

headers, and a collection of form variables if supplied. From this information, the library either issues an error message, serves data from preregistered files or directories, or passes control to a registered handler function. For handler functions, SWILL executes a user-defined callback function with the following prototype:

```
int handler(FILE *f, void *clientdata);
```

The first argument to the handler is a writable file object that the handler function may use to generate output. The second argument is a user-definable value that is supplied by the user when the handler function is first registered with SWILL. This argument is typically used to hold objects or other application specific data that pertains to the output being generated. For example, if an application had an integrated visualization capability, the clientdata argument might be used to refer to a specific visualization object and used in a handler function like this:

```
int
make_plot(FILE *f, void *clientdata) {
   Plot *p = (Plot *) clientdata;
   draw_image(p);    /* Make image */
   writepng(p,f);    /* Write PNG file */
}
...
main() {
  Plot  *ke;
  Plot  *vel;
  ...
  ke = KineticPlot();
  vel  = VelocityPlot();
  swill_register("ke.png", make_plot, ke);
  swill_register("vel.png", make_plot, vel);
}
```

When the handler function executes, all data written to its file object is collected and stored by SWILL. This data may include plain text, HTML, or binary data. When the handler completes its execution, this data is packaged into an appropriately formatted HTTP response and sent back to the browser–thus completing the request.

When the `swill_poll()` function services an incoming request, it does not return to the caller until the associated handler function has completed its execution. Thus, the handlers have complete control of the running simulation and full access to simulation data while they execute. The lack of concurrency also eliminates the need for complicated locking algorithms, assures us that only one handler will execute at once, and makes it safe for handler functions to perform complex operations such as sending messages to other nodes in a parallel machine.

6 I/O Handling

A central feature of SWILL is its I/O layer. When handler functions are invoked by the server, they are given a standard file object. This object can be passed to pre-existing I/O functions and used exactly like a normal file. The only difference is that when common I/O operations such as `fprintf()`, `fputs()`, `write()`, and `fwrite()` are applied to this object, their output is intercepted by SWILL and used to generate a web page.

To intercept common I/O operations, SWILL utilizes a feature of shared libraries that allows it to insert filter functions for common library calls[12, 9]. Replacements for the standard I/O functions are written and the `dlsym()` function is used to transparently pass control to the real implementation of each function as illustrated in Figure 3. To identify data intended for display on a web page, SWILL first opens a dummy file `/dev/null` for writing and passes it as the file parameter to the user defined handler functions. Then, when I/O operations are performed, the filter functions check for the use of this file object and redirects I/O to the web interface as necessary. In Figure 3, the `swill_file` variable contains a reference to a dummy file opened prior to the execution of the handler function. Then, in the implementation of `fwrite()`, a check is made against this file and I/O is redirected to a special SWILL specific function as shown. Otherwise, control is passed to the original `fwrite()` function in the C library.

Although this approach introduces an extra level of indirection to common I/O operations, it offers a number of advantages. First, the use of a dummy file object makes it easy to add SWILL to an existing application since it doesn't break the type system nor does it require existing code to be recompiled (applications must be relinked with the SWILL library however). In addition, if a program performs an unfiltered I/O operation (such as calling fcntl on the dummy file object), it has no dangerous side effects. Second, the use of `dlsym()` makes it possible to intercept common I/O operations without having to worry about their underlying implementation. This is important because libraries for supporting threads or parallel I/O often provide their own implementations of standard I/O functions. Provided that these libraries are packaged as

```
/* fwrite filter */
extern FILE *swill_file;   /* Reference to handler file */

ssize_t
fwrite(const void *buf, size_t size, size_t nitems, FILE *stream) {
    typedef ssize_t (*fwritetype)(const void *, size_t, size_t, FILE *);
    static fwritetype  real_fwrite = 0;

    /* Obtain the real implementation of fwrite() */
    /* This only executes once */
    if (!real_fwrite) {
       real_fwrite = (fwritetype) dlsym(RTLD_NEXT,"fwrite");
       assert(real_fwrite);
    }
    if (stream == swill_file) {
        /* Redirect I/O to the web page */
        return swill_fwrite(buf,size,nitems,stream);

    } else {
        /* Call the real fwrite function */
        return (*real_fwrite)(buf,size,nitems,stream);
    }
}
```

Figure 3: Example of a SWILL I/O filter

shared libraries and they appear after SWILL on the link line, the I/O filters will work regardless of the underlying implementation.

Currently, SWILL provides filters for the `printf()`, `fprintf()`, `fputs()`, `fputc()`, `fwrite()`, and `write()` functions in the C standard library. The implementation of these filters are similar to the code in Figure 3–adding filters for new functions is straightforward. In addition, the library allows output to standard output to be captured as illustrated in section 4. As for the performance impact of this approach, the use of a filter only introduces an extra function call and a handful of comparisons operators to each I/O operation. Given that applications tend to perform bulk I/O operations and that I/O operations often involve a trap to the operating system kernel, this extra overhead is negligible compared to the overall cost of performing I/O.

7 Handling of HTTP Metadata

One problem with supporting a web interface is that applications must deal with a variety of metadata related to the HTTP connection. This information includes content types, caching behavior, and access to form variables. There is very little than can be done to incorporate this aspect of the interface in the I/O model previously described. However, SWILL tries to hide as much of this complexity from the user as possible.

For example, content types are implicitly determined by file suffixes. Thus, a handler function for "cells.txt" implies that the resulting output will be plain text whereas "cells.html" implies HTML. Similarly, binary data for dynamically generated images can be specified using a suffix such as `*.jpg` or `*.png`. If it is ever necessary to explicitly access or set HTTP header fields, the following pair of functions can be used:

```
char *swill_getheader(char *name);
void swill_setheader(char *name, char *val);
```

Similarly, if a handler function wants to receive variables specified on an HTML form. The following functions are used:

```
char  *swill_getvar(char *name);
int    swill_getint(char *name);
double swill_getdouble(char *name);
```

Thus, if a scientist wanted to write a handler function that accepted input from a form, the code might look like this:

```
void
web_make_plot(FILE *f, void *clientdata) {
  Plot *p = (Plot *) clientdata;
  double xmin,xmax;
  double ymin,ymax;

  xmin = swill_getdouble("xmin");
  xmax = swill_getdouble("xmax");
  ymin = swill_getdouble("ymin");
  ymax = swill_getdouble("ymax");
  make_plot(p,xmin,xmax,ymin,ymax);
  writepng(p,f);
}
```

8 MPI Support

One of the most common techniques for writing parallel applications is to use a SPMD-style programming model where each processor executes the same sequence of operations, but with different data. In this case, the main simulation loop remains unchanged except individual functions such as `compute_forces()` and `integrate()` now utilize MPI functions.

SWILL uses a similar programming model in which all of its internal functions are modified to support parallelism, but the high level API remains unchanged. Thus, if a scientist wanted to use SWILL with an MPI program, it would work the same way as in previous examples. That is,

```
/* Initialize MPI */
MPI_Init(&argc, &argv);

/* Initialize SWILL */
swill_init(3737);
swill_handle("stdout>cells.txt",
               debug_cells,0);

for (i = 0; i < nsteps; i++) {
    compute_forces();
    integrate();
    boundary_conditions();
    redistribute_data();
    if (!(i % output_freq)) {
        write_output();
    }
    /* Check for connections */
    swill_poll();
}
```

Internally, parallelism is handled by modifying the behavior of the server and its connection protocol. First, although all nodes must call `swill_init()`, the function only opens a port for incoming connections on the master node (rank 0). Second, the `swill_poll()` function is turned into a global operation that must be performed by all nodes. In this case, the master node polls for a incoming connection and broadcasts a status to all of the other nodes. If no request is received, the master sends an empty request to the nodes and `swill_poll()` returns. If a request for a simple file is made or the master node encounters an HTTP error, it services the request on its own and broadcasts an empty request to the nodes. Otherwise, the master broadcasts the incoming HTTP request to all of the other nodes at which point an appropriate handler function is invoked. In this case, the handler function runs in parallel where it may use MPI functions, if necessary. Once all of the handler functions return, their output is gathered by the master and concatenated to form a response. This process is illustrated in Figure 4.

The implementation of SWILL's file serving capability is tailored to the SPMD model of parallel programming. Indeed, many applications that we have encountered are written in a manner that either takes advantage of a shared filesystem or bypasses the issue of where output files reside altogether, by passing all output to the master node and performing all file I/O there. There is an alternative pattern of filesystem usage, however, that, while not catered to explicitly by SWILL's fileserving capability, nonetheless does not present a problem to our implementation. This is the scenario of node-dependent file content, occasionally seen in Beowulf clusters. While there is no function for registering a file or directory in a way that will return concatenated copies of its contents from each node, this can be accomplished by registering a function that will perform that role. Thus, node-specific temporary files and log files can still be accessed via a registered function that outputs their content to a SWILL-accessible file on each or a subset of the nodes.

9 Use with Frameworks

An increasing number of scientists are migrating their applications to scripting languages and other advanced simulation environments. These environments tend to complicate the execution and I/O model of an application. Despite this, it is possi-

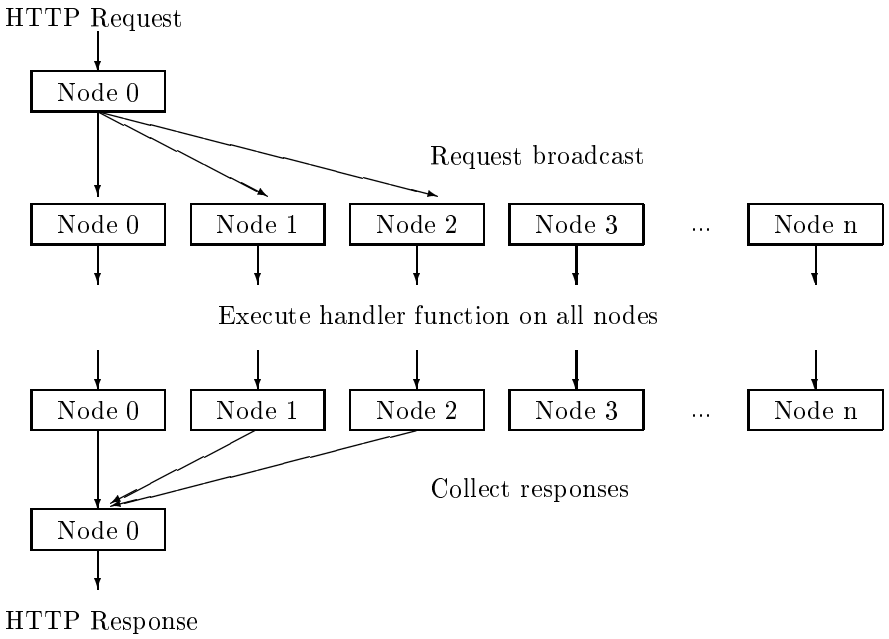

Figure 4: HTTP request handling with SWILL and MPI

ble to use SWILL in this setting. For instance, if a framework provides its own I/O abstraction layer, it may be possible to support SWILL as a new type of I/O device. Alternatively, it may be possible to rely on the I/O filtering approach described in this paper.

As an example, consider an application built as an extension to Python, a popular scripting language used in a wide variety of scientific applications [13]. If SWILL is linked to a dynamically loadable Python extension module, the SWILL I/O filters are still applied to the I/O procedures used by that module. Thus, it is still possible to use the web interface as before. A much more interesting situation arises if the Python interpreter itself is linked with SWILL. In this case, I/O performed by the Python interpreter as well as I/O performed by dynamically loadable modules can be routed through the filters. In this case, it is possible to generate web pages from I/O operations occurring in multiple languages. Furthermore, if a scripting interface to SWILL is built, it would be possible to implement the handler functions as script functions that execute a mix of interpreted and compiled procedures.

10 Discussion

The primary distinction between SWILL and previous approaches to the web monitoring problem is SWILL's layering of the web-content generation onto preexisting I/O functions. However, this approach requires applications to use a fairly standard I/O programming model. If a program uses a highly specialized or non-standard I/O API, SWILL may not be able to capture its I/O operations. For instance, it may not be possible to capture operations performed through a parallel I/O library. On the other hand, the files generated by such a library are unlikely to be of much interest through a web interface (i.e., it is unlikely that a scientist would actually want to download a huge datafile through their web browser on a remote machine).

Another aspect of the implementation is that the I/O filtering process requires the real implementation of the I/O functions to be contained in a shared library. In practice, this is not an issue because virtually all modern machines provide shared library support and bundle common system libraries as shared objects[1].

On the subject of performance, one major concern

[1] There is a cult of scientists that insist on compiling their applications as static executables to achieve slight performance improvements. In reality, the performance gains of static linking are marginal and there is very little reason to do this, given the degree to which it restricts application flexibility and extensibility.

caused by this approach is the performance impact the web interface might have on large parallel systems. In particular, each invocation of the polling function minimally involves a blocking broadcast from the master node. Clearly, it would be a bad idea for an application to spin on the polling function or to issue a polling operation every few microseconds. A much more realistic implementation might poll after every few timesteps of a simulation. In this case, the polling operation might only execute every few seconds–a process that is not going to add a significant performance overhead to most applications. It is also important to note that SWILL is primarily intended to support infrequent web access (i.e., a scientist periodically checking on their simulation). Because of this, SWILL would not be appropriate as a means for servicing thousands of requests during program execution. Similarly, the library is not designed to support bulk data transfer of huge files through a high speed HTTP connection.

Finally, it is important to note that the use of a web interface introduces a variety of considerations related to security, networking, and reliability. Since SWILL relies on polling, special care needs to be given to avoid network connectivity problems. For instance, a user would not want a bad network connection to freeze the simulation in an infinite I/O wait state. Similarly, a user certainly wouldn't want their application to come under attack from an outsider. To deal with these situations, the library can apply timeouts to recover from dead connections. In addition, it is possible to apply basic user authentication and IP address filtering to incoming connections[2]

11 Related Work

A substantial number of research projects have previously addressed the problem of simulation monitoring and remote control. Much of this work can be found in the general area of computational steering [11, 16]. With computational steering systems, the primary focus is on adding fine-grained control and interactivity to scientific systems. To do this, applications are often instrumented with access points, built around dataflow systems, or implemented on top of a complex middleware layer [14, 15]. In certain cases, a scripting language interface has been used to provide interactive access [7, 3]. A common aspect of these systems is that the interactive interface tends to be tightly coupled to the underlying application. Furthermore, a lot of this work tends to focus on high-end interactive visualization such as that found in virtual environments. If a steering system provides remote simulation access, it is often achieved through the use of special network protocols or a complex middleware layer– thus requiring the use of special client software on the user's machine.

Research in distributed computing and computational grids have also addressed the problem of remote access to simulations. For example, systems such as Globus and Legion provide mechanisms for controlling and manipulating remotely running applications [8, 5]. However, much of this work tends to focus on the problem of coordinating heterogeneous computing resources rather than the problem of monitoring application specific simulation data. In certain cases, web access has been added as a feature to various application frameworks. For example, Cactus provides a web interface module that allows certain parts of a simulation to be monitored and controlled [6].

12 Current Status

Currently, SWILL is written to support applications written in ANSI C. The implementation contains approximately 3000 semicolons, most of which are related to the parsing and handling of HTTP requests. MPI support is isolated to a single module responsible for broadcasting requests and collecting responses. The I/O filters are isolated to a single module and require the use of the dynamic loader and shared libraries. If necessary, SWILL can be used without the use of I/O filter functions. However, its ease of use is diminished in this configuration.

In the future, it may be possible to extend the I/O filters and the handler function mechanism to support programs written in Fortran and C++. For C++, it might be possible to define a new IOStream class that allows I/O operations to be transparently redirected to the SWILL library. For Fortran, one would need to capture a different set of I/O operations and provide an alternative calling convention for the handler functions.

[2] No privileged system resources or daemons are required by SWILL. Hence, the application server runs with user permissions. Of course, if security is more important than science, a firewall can be used to restrict all access (except for when giving a Supercomputing demo).

13 Conclusions and Availability

The use of an embedded web server is a simple and effective way to provide remote access to long running scientific simulations. SWILL makes it easy to add this capability to existing software by allowing web pages to be dynamically generated from existing I/O functionality. In addition, SWILL's support for MPI simplifies the task of interacting with simulations running on large parallel machines and clusters.

SWILL is freely available under a GPL license. More information can be obtained at

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

14 Acknowledgments

Much of the early work on SWILL was derived from a web server interface built for the SPaSM molecular dynamics code at Los Alamos National Laboratory. SPaSM is currently maintained by Peter Lomdahl and Tim Germann in the theoretical physics division. In addition, we acknowledge Mike Sliczniak for his early contributions to the SWILL project.

References

[1] G. Allen, W. Benger, T. Goodale, et al, *Cactus Grid Computing: Review of Current Development*, Submitted to European Conference on Parallel Computing (Euro-Par) (2001), Manchester 28-31.

[2] 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.

[3] D.M. Beazley and P.S. Lomdahl, *Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations,* in Proceedings of Supercomputing'96, (1996).

[4] T. Boutell, *gd: A Graphics Library for Fast Image Creation*, `http://www.boutell.com/gd`

[5] S. Chapin, J. Karpovich, A. Grimshaw, *The Legion Resource Management System*, JSSPP'99, San Juan, Puerto Rico, (1999).

[6] T. Dramlitsch, G. Allen, E. Seidel, *Efficient Techniques for Distributed Computing*, Submitted to the Tenth IEEE International Symposium on High Performance Distributed Computing (HPDC10), San Francisco, CA. (2001).

[7] Dubois, P.F., *Making Applications Programmable*, Computers in Physics 8, 1 (1994). p. 70-73.

[8] I. Foster, C. Kesselman, S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, (to be published in Intl. J. Supercomputer Applications, 2001).

[9] R.A. Gingell, M.L. Xuong, T. Dang, M.S. Weeks, *Shared Libraries in SunOS*, USENIX Summer Conference. (1987).

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

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

[12] J. R. Levine, *Linkers & Loaders.* Morgan Kaufmann Publishers, 2000.

[13] M. Lutz, *Programming Python, 2nd Ed.*, O'Reilly & Associates, (2001).

[14] M. M. Muralidhar, *Discover: An Environment for Web-based Interaction and Steering of High-Performance Scientific Applications*, Concurrency-Practice And Experience Concurrency. (2000).

[15] S.G. Parker and C.R. Johnson, *SCIRun: A Scientific Programming Environment for Computational Steering*, In Proceedings of Supercomputing'95, (1995).

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