Designing a Runtime System for Volunteer Computing

David P. Anderson[1]
Carl Christensen[2]
Bruce Allen[3]

[1] UC Berkeley Space Sciences Laboratory
[2] Dept. of Physics, University of Oxford
[3] Physics Dept., University of Wisconsin - Milwaukee

Abstract

Volunteer computing is a form of distributed computing in which the general public volunteers processing and storage to scientific research projects. BOINC, a middleware system for volunteer computing, is currently used by about 20 projects, to which over 600,000 volunteers and 1,000,000 computers supply 350 TeraFLOPS of processing power. A BOINC client program runs on the volunteered hosts and manages the execution of applications. Together with a library linked to applications, it implements a **runtime system** providing process management, graphics control, checkpointing, file access, and other functions. This runtime system must handle widely varying applications, must provide features and properties desired by volunteers, and must work on many platforms. This paper describes the problems in designing a runtime system having these properties, and how these problems are solved in BOINC.

1. Introduction

1.1 Volunteer computing and BOINC

Volunteer computing is a form of distributed computing in which the general public volunteers processing and storage resources to scientific research projects. Early volunteer computing projects include the Great Internet Mersenne Prime Search [9], SETI@home [1], Distributed.net [6] and Folding@home [10]. Today the approach is being used in many areas, including high-energy physics, molecular biology, medicine, astrophysics, and climate dynamics. This type of computing can provide great power (SETI@home, for example, has accumulated 2.5 million years of CPU time in 7 years of operation). However, it requires attracting and retaining volunteers, which places many demands both on projects and on the underlying technology.

BOINC (Berkeley Open Infrastructure for Network Computing) is a middleware system for volunteer computing [2]. BOINC is being used by a number of projects, including SETI@home, Climateprediction.net [5], LHC@home [11], and Einstein@Home [7]. Volunteers participate by running BOINC client software on their computers (or **hosts**). They can attach each host to any set of projects, and can control the allocation of resources among projects.

A BOINC-based project provides its own servers. Hosts download application executables and data files from servers, carry out **tasks** (by running applications against specific data files), and upload the output files. BOINC software includes server-side components, such as scheduler and daemon programs that manage task distribution and collection [3], and web-based interfaces for volunteers and project administrators. This paper, however, is concerned only with the BOINC client software.

1.2 BOINC design goals

Many of BOINC's design goals arise from the need to attract and retain volunteers. We consider several factors in this area to be critical:

● Volunteers should, by default, not perceive significant system slowdown, or exhaustion of disk space, as a result of BOINC's activity. However, some volunteers place a high priority on productivity. These volunteers want control of when and how their computing resources are used – to choose, in effect, the tradeoff between invisibility and productivity. BOINC lets volunteers specify **preferences**, such as the hours during which computation can be done, whether to compute only during idle periods, and how many CPUs (on a multiprocessor system) can be used.
● Volunteers must be given incentives to participate. What constitutes an incentive varies between volunteers. Some enjoy seeing application graphics, either in a window or as a screensaver. Others are motivated by competition with respect to computer speed and donated CPU time; BOINC must therefore make accurate and cheat-resistant measurements of the work done by each host. This data is used by independent web sites that show lists of top users and teams.
● The client software must be simple to install, maintain, and use. The software should be **autonomic**: it should recover without volunteer intervention from problems (even those caused by unexpected volunteer actions).

Another design goal is to handle widely differing applications and tasks. For example, a SETI@home tasks takes about 2 CPU hours on a typical PC (3 GHz Pentium) and uses 20 MB RAM, while a climate-modeling task may take several months of CPU time and hundreds of MB of RAM and disk space. Applications may consist of a single process or a pipeline of programs sequenced by a separate coordinator. Some applications are not CPU-intensive and must be scheduled accordingly.

Other design goals include support for application debugging (by making diagnostic information available to project developers), and support for many platforms (such as Windows, Macintosh OS X, OS/2 and UNIX).

1.2 The BOINC runtime system

The BOINC client software consists of several components (see Figure 1):

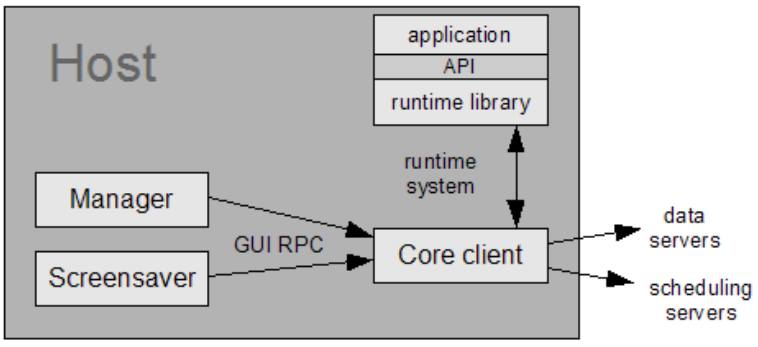

Figure 1: the BOINC client software includes a 'core client' that executes applications and interacts with them through a runtime system.

• **Applications** are typically long-running scientific programs. They may consist of a single process or a dynamic set of multiple processes (see Section 2.3).
• The BOINC **core client** program communicates with schedulers, uploads and downloads files, and executes and coordinates applications.
• The BOINC **Manager** provides a graphical interface allowing users to view and control computation status (see Figure 2). For each task, it shows the fraction done and the estimated time to completion, and lets the user open a window showing the application's graphics. It communicates with the core client using remote procedure calls over TCP.
• A BOINC **screensaver** (if enabled by the volunteer) runs when the computer is idle. It doesn't generate screensaver graphics itself, but rather communicates with the core client, requesting that one of the running applications display full-screen graphics.

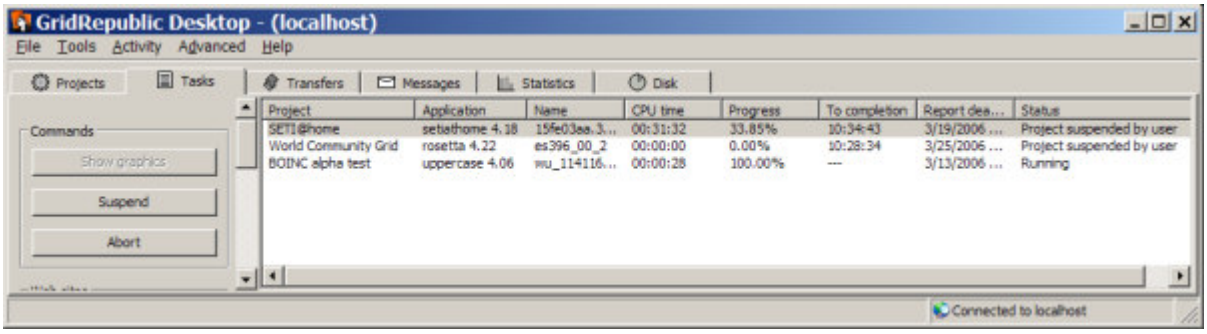

Figure 2: The BOINC Manager provides a graphical user interface.

The core client schedules applications. On a multiprocessor with *n* CPUs, it attempts (if user preferences allow it) to run *n* applications at once. It does preemptive round robin scheduling among applications, so that volunteers who participate in multiple projects see periodic change. Applications that are preempted may be suspended or forced to exit, depending on volunteer preferences. The core client also guards against certain types of application misbehavior. Each task has project-specified limits on memory usage, disk usage, and computation (number of floating-point operations). The runtime system periodically measures these quantities and reports them to the core client. If the limits are exceeded, the core client aborts the application.

The core client interacts with applications in various ways. This interaction is implemented by a **runtime system** implemented by the core client and a **runtime library** linked with applications. Most of this interaction is hidden from the application developer. In cases where the developer needs to be involved, this is done via a BOINC **Application Programming Interface** (API).

This paper describes the problems and design decisions we encountered in developing the BOINC runtime system, and our solutions to these problems. Section 2 describes the overall architecture of the runtime system. Sections 3 through 11 discuss the functional areas provided by the runtime system in terms of requirements, problems and solutions. Sections 12 and 13 discuss related work and give conclusions.

2. Overall architecture

2.1 Shared-memory message-passing

The runtime system requires bidirectional communication between the core client and applications. How should this work? Operating systems offer a variety of mechanisms for interprocess communication, process control, and synchronization. For example, POSIX-compliant systems have signals, semaphores, and pipes. Windows has mutexes, messages, and various system calls for process and thread control. We avoided platform-specific mechanisms because of the resulting code complexity.

Instead, the BOINC runtime system is based on **shared-memory message passing**. For each application it executes, the core client creates a shared memory segment containing a data structure with a number of unidirectional message channels. Each channel consists of a fixed-size buffer and a 'present' flag. Message queuing, if needed, is provided at a higher software level. All messages are XML, minimizing versioning problems.

The BOINC runtime system uses eight message channels, four in each direction. For example, one channel carries task control messages (telling the application to suspend, resume, quit or abort) while another conveys graphics-related messages (telling the application to create or destroy graphics windows).

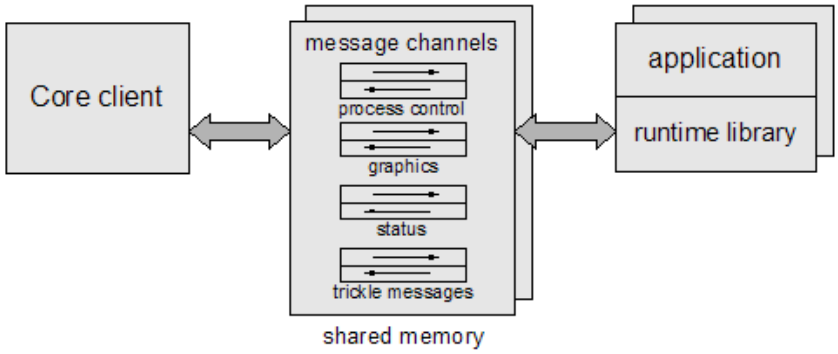

Figure 3: The core client communicates with applications by shared-memory message passing

Compared with alternatives (such as pipes, sockets, or signals) shared-memory message passing has significant advantages. First, all operating systems supported by BOINC have shared-memory facilities with similar semantics. Secondly, handling compound applications (see Section 2.3) is simpler; each program in the application is typically interested in a subset of the message channels. For example, the graphics program monitors the graphics channel, while the coordinator monitors the process control channel.

2.2. Application thread structure

The runtime system requires that applications perform various activities in addition to their main computation. BOINC uses **threads** for this purpose (pthreads on Unix, native threads on Windows). The thread structure in Unix is as follows (see Figure 3).

● A **worker thread** executes the application's main computation and handles a 1 Hz SIGALRM signal, which has two purposes. First, since pthreads does not allows one thread to get the CPU time of another thread, the handler calls `getrusage()` and stores the result in a variable. Second,

since pthreads does not support suspend/resume between threads, the handler simulates suspension by sleeping if a flag is set.

- A **graphics thread** executes a GUI event loop and calls the application's rendering function (see Section 8).
- A **timer thread** executes a timer function once per second. This performs periodic actions handling process control messages (see below). This is done in a separate thread (rather than in the SIGALRM handler) because it uses C library functions (such as `printf()`) that cannot safely be called from an asynchronous signal handler.

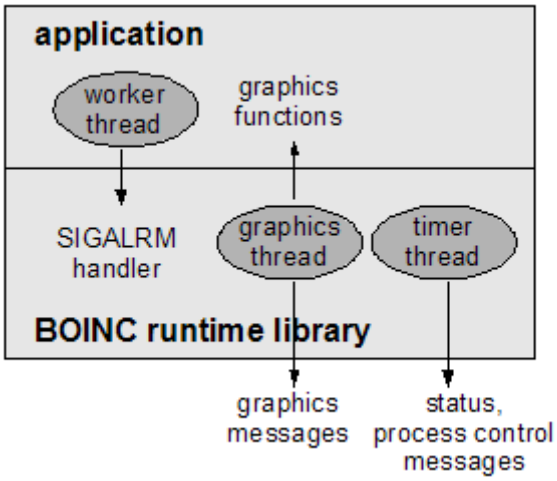

Figure 3: The BOINC runtime system's thread structure

In Windows, the thread structure is similar, except that the timer thread and SIGALRM handler are replaced by a periodic 'multimedia timer' that runs in a thread provided by the C runtime library.

2.3. Simple and compound applications

BOINC supports both **simple** and **compound** applications. Simple applications consist of a single program, and their scientific code, graphics code, and the BOINC runtime library reside and execute in a single address space. Compound applications consist of several programs – typically a **coordinator** that executes one or more **worker programs**. The coordinator, for example, might run pre-processing, main, and post-processing programs in sequence, or it might launch one or more programs (e.g. coupled climate models) that run concurrently and communicate via shared memory. It might run a graphics program concurrently with a scientific program.

The BOINC runtime library is linked with each program of a compound application. The BOINC API lets each program specify which message channels it will handle, and whether the message handling should be done by the runtime system or by the application. In the example shown in Figure 4, the coordinator handles process control messages, while the graphics program handles graphics messages.

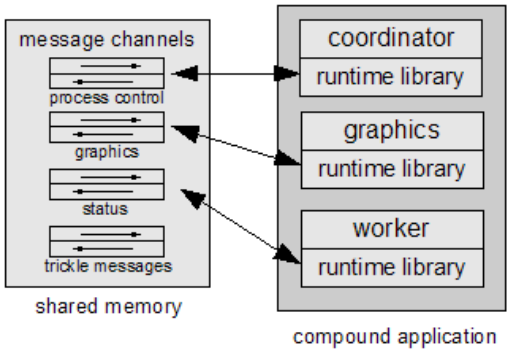

Figure 4: A compound application consists of several processes, each of which handles particular message channels

3. Task management

3.1 Task control

The runtime system must allow the core client to perform various control operations on running tasks:

- **Suspend**: temporarily stop the main computation (i.e., stop the worker thread; the graphics thread must continue normally, otherwise the graphics window would freeze).
- **Resume**: undo suspend.
- **Quit**: have the application exit all processes, to be restarted later.
- **Abort**: have the application exit all processes, not to be restarted later.

Suspend, resume, and quit are used for CPU scheduling, for automatic suspension due to preferences, and for suspension in response to GUI requests. Abort is used to enforce user requests and to kill runaway applications.

These operations are implemented by sending messages to the process control channel. In Unix this channel is monitored by the timer thread. On receiving a suspend message, it sets a flag which is checked by the SIGALRM handler. If the flag is set, the handler repeatedly sleeps for a second (thus suspending the worker thread, which handles SIGALRM). In Windows, the channel is monitored by the timer event, which suspends the worker thread using `SuspendThread`().

The coordinator of a compound application typically handles process control messages. In this way, for example, the worker programs can be shut down cleanly, e.g. shared-memory segments can be detached and closed before quitting.

If an application doesn't respond to a quit or abort request within a few seconds, the core client forcibly kills the application and sub-processes via an OS-specific mechanism (a KILL signal in Unix and `TerminateProcess`() in Windows).

3.2 Orphaned and duplicate processes

Sometimes the core client exits unexpectedly (for example, because it crashes). In these situations, a mechanism is needed that will cause applications to eventually exit. BOINC uses

heartbeat messages, which are sent once per second from the core client to each application. If an application doesn't get a heartbeat message for 30 seconds, it exits.

Each application executes in a directory containing its input and output files (see Section 6). To prevent duplicate copies of an application from executing in the same directory, the runtime system uses a lock file. The API initialization routine tries to acquire the lock file; if it can't, it waits for 30 seconds (allowing the heartbeat mechanism to take effect) and tries again.

3.3 Reliable termination

The core client uses standard functions (such as `waitpid()` on Unix) to find when applications have finished and whether they exited normally. On some versions of Windows, when a program is killed externally by the user, it is indistinguishable (from the core client's viewpoint) from a call to `exit(0)`. To solve this problem, the BOINC API finalization routine writes a 'finished file'. If the core client detects that a program has exited unexpectedly but no 'finished file' is found, it restarts the application.

5. Status and credit reporting

5.1 CPU time and VM usage

The core client needs to know the current CPU time and memory usage of each application every second or so. The CPU time should include only the worker thread, not the graphics thread. Some platforms (e.g. UNIX) don't provide a mechanism that lets the core client obtain these data directly. Instead, the BOINC runtime system makes the measurements (as described in Section 2.2) and reports them to the core client through the 'status' message channel.

5.2 Fraction done

The core client needs an estimate of the application's fractional task completion (0 to 1) every second. This is used for scheduling purposes (e.g. to decide when to fetch more work) and is shown to the user in the GUI. For this purpose, the application calls

```
boinc_fraction_done(double);
```

to report its current fractional task completion. This function should be called at least once per second. In compound applications involving programs run in sequence, the coordinator program is responsible for dividing the overall fraction done among the programs. The fraction done is reported to the core client through the 'status' message channel.

5.3 Credit reporting

BOINC keeps track of the work done by each host and each volunteer. This is used to display web-based leader boards, and is also shown in the BOINC manager. The notion of "work" includes the number of floating-point and integer operations performed. By default this is estimated by multiplying benchmark scores by the application's CPU time. However, this estimate is erroneous in some cases (e.g. if the application is memory-intensive, and the host has a slow memory system relative to its CPU speed). To improve the accuracy of credit estimates, the BOINC API includes functions

```
boinc_ops_per_cpu_sec(double floating_point, double integer);
boinc_ops_cumulative(double floating_point, double integer);
```

These functions let the application report the results of an internal benchmark, or a direct estimate of the number of operations done. This information is conveyed via messages to the core client, which in turns sends it project servers where credit is computed and granted.

6. Directory structure and file access

The directory structure used by the BOINC runtime system has several requirements. BOINC must run tasks in separate directories, so that their temporary files do not conflict. At the same time, if two concurrent tasks use a particular (read-only) file, there should be only a single copy of the file. Finally, minimal changes to the source code of legacy applications should be needed. BOINC's design is as follows (see Figure 5).

- For each project to which the host is attached, a directory is created that contains all files (input, output, executable) for that project.
- For each active task (executing or preempted) a directory (or **slot**) is created, in which the task executes. Slot directories typically do not contain data files, but rather 'link files', which are analogous to UNIX symbolic links (Windows does not have symbolic links).

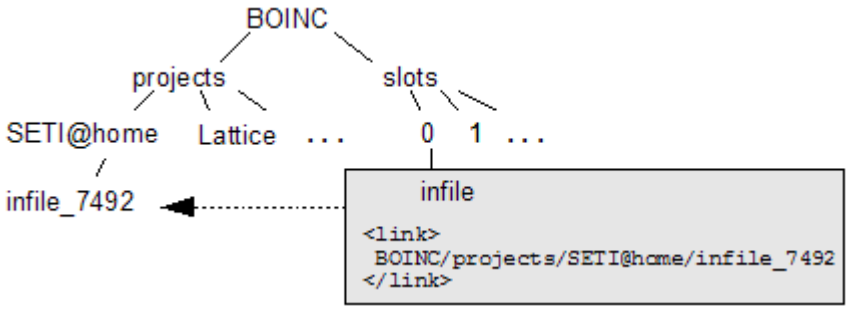

Figure 5: Each task runs in a 'slot directory' with link files that map logical names ('infile') to physical names ('infile_7492').

To access files, an application calls a function that maps the logical file name to a physical file name (by parsing the link file). For example, instead of

```
f = fopen("infile", "r");
```

an application might use

```
boinc_resolve_filename("infile", physical_name);
f = boinc_fopen(physical_name, "r");
```

`boinc_fopen()` emulates `fopen()` and deals with platform-specific problems. On Windows, where virus-checking and indexing programs can briefly lock files, it does several retries at 1-second intervals. On Unix, where signals can cause `fopen()` to fail with EINTR, it checks for this and does a few retries.

7. Checkpointing

BOINC expects applications to do checkpoint/restart, so that they can quit and restart repeatedly and still finish their intended computation. BOINC user preferences include a minimum interval between periods of disk activity. This is useful for laptops whose disks spin down to conserve power. The BOINC runtime system must allow applications to checkpoint frequently (to minimize wasted CPU time) but must respect the minimum disk interval.

BOINC applications typically have particular points in their execution where the state of the computation can be represented compactly (e.g. by the values of outer loop indices). These "checkpointable states" may be separated by milliseconds or by minutes. The BOINC API provides a function

```
bool boinc_time_to_checkpoint();
```

that should be called whenever the application is in a checkpointable state. It can be called frequently (hundreds or thousands of times a second). It returns true if the minimum disk interval has elapsed since the last checkpoint. If so, the application should write a checkpoint file and call

```
boinc_checkpoint_completed();
```

These functions automatically make checkpointing a critical section with respect to quit messages (see Section 3.1). They also inform the core client when the application has checkpointed, so that it can correctly account total CPU time, and so that it can avoid doing preempt-by-quit for applications that haven't checkpointed recently.

7.1 Output file integrity

Many BOINC applications write incrementally to output files. If an application is preempted by quitting at a time when has extended an output file since the last checkpoint, the same output will be written when the task runs again, producing an erroneous output file. There are several ways of dealing with this. The application can copy output files during checkpoint; this is potentially inefficient. It can store the size of output files in the checkpoint file, and seek to these offsets on restart. Or it can use a set of `printf()`-replacement functions (supplied by BOINC) that buffer output in memory, and flush these buffers during checkpoint.

8. Graphics

SETI@home set a high standard for volunteer computing applications by providing an animated visualization of the current computation, viewable either as a screensaver or in a window. BOINC allows application developers to produce similar graphics without writing a lot of code. An application can provide graphics simply by supplying a function

```
app_graphics_render();
```

that draws a frame using OpenGL primitives [17], and calling an initialization function that creates a graphics thread. This thread executes code (in the BOINC runtime library) that monitors the graphics message queue and that implements a window-system event handling loop. Requests to open and close windows originate from the BOINC Manager and screensaver, and are relayed to applications by the core client. The runtime library handles the creation and destruction of windows (using native calls on Windows, and GLUT on other platforms). It calls application-supplied graphics functions (for rendering and input handling) as needed. It implements a

throttling mechanism that limits the fraction of CPU time used by the graphics thread. Typically it calls the render function 10 times a second, but on hosts without graphics acceleration this will be reduced to one frame per second or so.

Applications must link dynamically to system graphics libraries such as OpenGL and X11; otherwise rendering won't be hardware-accelerated. However, if an executable that dynamically links a library is run on a system that doesn't have the library, it will fail to start. Some Unix systems have no graphics libraries. BOINC solves this problem by dividing the application into a main program and a shared library containing the application's graphics code. The shared library dynamically links system graphics libraries, but the main program does not. BOINC's graphics-initialization function attempts to load the shared library; if system graphics libraries are missing, this will fail, in which case the main program continues to execute, but without graphics. This mechanism is not needed on Windows or Mac OS X, where system graphics libraries always exist.

In the above approach, the worker and graphics thread run in the same address space. This makes it easy to show a detailed, up-to-the-second view of the scientific calculation. For some applications, this is not important; it may be easier to implement graphics as a separate program that uses information in output or checkpoint files. The BOINC API supports this.

9. Remote diagnostics and debugging

Applications can fail by crashing or going into infinite loops. Some failures occur only in specific contexts – CPU type, OS version, library version, even CPU speed. Such failures may be common on volunteer hosts, yet never occur on the project's development machines. The BOINC runtime system has several features that collect failure information:

- An application's standard error output is directed to a file and returned to the project's server for all tasks, failed and not.
- If an application crashes, stack trace is written to standard error. It the application includes a symbol table, the stack trace is symbolic.
- If an application is aborted (because the task exceeds time, disk, or memory limits, or is aborted by the user) a stack trace is written to standard error.

All information about a task (exit code, signal number, standard error output, volunteer host platform) is stored in a relational database on the server, making it easy to isolate the contexts in which failures occur. Many BOINC-based projects have small "alpha testing" projects, with enough volunteers to cover the main platforms, so that context-specific problems can be fixed before applications are released to the public.

10. Long-running applications

Some BOINC applications have extremely long tasks. For example, Climateprediction.net tasks can take several months or more to run a single climate model [18]. These long-running applications require mechanisms for communicating with project servers on a regular and moderately frequent basis.

10.1 Trickle messages

Trickle messages let applications communicate with the server during the execution of a task. Messages may go in either direction: 'trickle up' messages go from application to server, 'trickle down' messages go from server to application. Typical uses of this mechanism include:

- The application sends trickle-up messages containing its current CPU usage, so that users can be granted incremental credit (rather than waiting until the end of the task).
- The application sends a trickle-up message containing a summary of the computational state, so that server logic can decide if the computation should be terminated.
- The server sends a trickle-down message telling the application to terminate.

Trickle messages are asynchronous and reliable. Trickle messages are conveyed in scheduler RPC messages, so they may not be delivered immediately after being generated by the application. The BOINC API includes functions for sending and receiving trickle messages.

10.2 Intermediate file upload

Long-running applications may need to upload particular output files before the task is finished, so that the project can see intermediate results even if a task is never completed. The BOINC API includes functions to indicate that an output file is complete and should be uploaded, and to poll the upload status of such a file.

11. Non-CPU-intensive applications

Some BOINC applications are not CPU intensive. These applications run whenever BOINC runs, but have short and infrequent periods of activity, sleeping the rest of the time. Examples include:

- Applications that study network structure and performance. When active, they establish one or more network connections for a short period, measuring the latency and/or bandwidth on these connections, or counting the number of hops to 'landmark' hosts.
- Applications that study the dynamics of computer usage. These applications typically run periodically, and measure system metrics such as CPU load, memory usage, and I/O rates.
- Applications that provide a network service. These applications accept network connections and perform operations (such as relaying messages or accessing files) that involves little CPU time.

BOINC allows tasks to be marked as **non-CPU-intensive**. Such tasks are treated specially. The core client always runs the corresponding application. The BOINC API supplies functions that suspend and resume all BOINC activities other than the caller; this allows accurate system measurements to be made.

12. Related work

Several other middleware systems for volunteer computing have been developed. These systems have had different requirements and assumptions than has BOINC, and their runtime systems differ accordingly. For example, Bayanihan [13] required that applications be written in Java, allowing the use of the widely-available Java runtime system. This was not acceptable for BOINC, which supports applications in various languages including FORTRAN. Entropia emphasized security, and developed a runtime system (on Windows only) that sandboxes applications by trapping all system calls. This level of protection is not provided by BOINC.

A variant of volunteer computing is provided by Condor [12, 15], which was designed for use on resource pools such as groups of departmental workstations. The Condor runtime system supports process migration; if a user returns to her workstation, the job running there is migrated to an idle workstation. Condor also uses a remote system call technology that traps library calls and sends them over the network to machine where job was submitted.

13. Conclusion

Volunteer computing offers a unique challenge for middleware designers. Large, complex science applications must be run (often sporadically) on volunteer PCs. Some volunteers want fancy graphics and total manual control; others want the system to stay completely out of their way. At the heart of such middleware is the runtime system – the kprotocol between the science applications and the controlling software (the core client, in the case of BOINC). We have described the various requirements, problems, and solutions that have gone into the BOINC runtime system.

BOINC has succeeded in handling a wide range of applications, running on all common platforms, and attracting a large volunteer population (about 500,000 at last count). The software is small (the runtime system is about 5,000 lines, and the core client is about 20,000 lines) and easy to port and maintain. We attribute this in part to limited goals: for example, BOINC doesn't support binary compatibility (applications require source modification and recompilation), it doesn't provide sandboxing, and it doesn't support process migration.

BOINC is being actively developed. We plan to add a limited form of sandboxing in which applications run as an unprivileged user. We plan to support graphics coprocessors for numerical processing. On a host with a GPU, the core client will treat the GPU as a schedulable resource, and will run GPU-capable applications concurrently with applications that primarily use the CPU. We expect that BOINC will eventually be useful not only for volunteer computing but also for "desktop grid" computing within organizations.

This work was supported by the National Science Foundation under grants SCI-0221529 and SCI-0438443. Thanks to Rom Walton, Bernd Machenschalk, Eric Korpela, and Charlie Fenton, who contributed ideas, discussion, and code to the BOINC runtime system.

References

[1] D.P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, D. Werthimer. "SETI@home: An Experiment in Public-Resource Computing". *Communications of the ACM*, 45(11), November 2002, 56-61.

[2] D.P. Anderson. "BOINC: A System for Public-Resource Computing and Storage". 5th IEEE/ACM International Workshop on Grid Computing, Nov. 8 2004, Pittsburgh, PA. 365-372.

[3] D.P. Anderson, E. Korpela, and R. Walton. "High-Performance Task Distribution for Volunteer Computing". First IEEE International Conference on e-Science and Grid Technologies, 5-8 December 2005, Melbourne.

[4] A. Chien, B. Calder, S. Elbert, and K. Bhatia. "Entropia: architecture and performance of an enterprise desktop grid system". J. Parallel Distrib. Comput. 63(2003) 597-610.

[5] C. Christensen, T. Aina and D. Stainforth. “The Challenge of Volunteer Computing With Lengthy Climate Model Simulation”. 1st IEEE International Conference on e-Science and Grid Computing, Melbourne, Dec 5-8 2005.
[6] Distributed.net, http://distributed.net
[7] Einstein@Home, http://einstein.phys.uwm.edu/
[8] C. Germain, V Neri, G. Fedak and F. Cappello. “XtremWeb: Building an Experimental Platform for Global Computing”. First IEEE/ACM International Workshop on Grid Computing. December 17-20, 2000, Bangalore, India.
[9] GIMPS, http://www.mersenne.org/prime.htm
[10] S.M. Larson, C.D. Snow, M. Shirts and V.S. Pande. “Folding@Home and Genome@Home: Using distributed computing to tackle previously intractible problems in computational biology”. *Computational Genomics*, Horizon Press, 2002.
[11] LHC@home, http://athome.web.cern.ch/athome/
[12] M.J. Litzkow, M. Livny, M.W. Mutka. “Condor - A Hunter of Idle Workstations”. Proceedings of the 8th International Conference of Distributed Computing Systems, pages 104-111, June, 1988.
[13] L.F.G. Sarmenta, “Bayanihan: Web-Based Volunteer Computing Using Java”. Lecture Notes in Computer Science 1368, Springer-Verlag, 1998. pp. 444-461.
[14] D. A. Stainforth, T. Aina, C. Christensen, M. Collins, N. Faull, D. J. Frame, J. A. Kettleborough, S. Knight, A. Martin, J. M. Murphy, C. Piani, D. Sexton, L. A. Smith, R. A. Spicer, A. J. Thorpe & M. R. Allen, Uncertainty in predictions of the climate response to rising levels of greenhouse gases, Nature, 433, pp.403-406, January 2005.
[15] Douglas Thain, Todd Tannenbaum, and Miron Livny, "Distributed Computing in Practice: The Condor Experience" Concurrency and Computation: Practice and Experience, Vol. 17, No. 2-4, pages 323-356, February-April, 2005.