SOS: Sonify Your Operating System

Andy M. Sarroff[1], Phillip Hermans[2], and Sergey Bratus[1]

[1] Department of Computer Science
[2] Department of Music
Dartmouth College, Hanover NH 03755, USA
`sarroff@cs.dartmouth.edu`

Abstract. The modern personal computer is capable of executing many millions of instructions per second. Some of these instructions may cause the computer to slow down, expose security vulnerabilities to the outside world, or cause other undesirable behavior. It is impractical and unwieldy to regularly probe the operating system's kernel and analyze enormous text files for detecting anomalous behavior. Yet the operating system is a real-time agent that often exhibits identifiable activity patterns. Under the right conditions humans are good at detecting temporal-acoustic patterns. We therefore suggest a framework for sonifying important aspects of the operating system. Our architecture is lightweight, generalizable to Unix-like systems, and open sourced. The system is built using the ChucK sound synthesis engine and DTrace dynamic tracing library yielding musical expressivity over operating system sonification.

Keywords: Sonification, operating systems, DTrace, ChucK, audio synthesis.

1 Introduction

A modern personal computer is capable of executing many millions of instructions per second. While a user is casually browsing the internet, listening to music, or writing a paper on their PC, the operating system kernel is managing a continuous stream of instructions. Some of these include manipulating files (e.g. opening, closing, reading, writing to files); controlling processes (e.g. loading an application into memory); and device management (e.g. mounting a hard drive). When a system is overwhelmed with instructions or when its memory is over-utilized, a user may experience diminished responsiveness. Such situations may be frustrating, especially when the user is unaware of the causes for unresponsiveness.

Whenever we are connected to a network, our PCs are constantly sending and receiving messages from other computers. Sometimes we are explicitly aware of this behavior. For instance when we browse the internet or check our email, we are aware that our computer is connecting to the outside world. However oftentimes our computers continue to send and receive data without our knowledge. Our computer may be downloading large files from the "mother ship" (e.g.

when the Mac OS downloads a system update from Apple's servers). Or we may have unwittingly left open a file sharing port that others are exploiting.

These are just a few scenarios in which important events are occurring in the background of our computing environment. Such events may negatively impact our interaction with the computer or worse, expose our personal data to the world. It behooves us to have some awareness of the automatic processes occurring regularly without our knowledge. Yet the instructions executed by the operating system at any given time, most of which are benign, are unwieldy in number.

There are many extant tools to analyze the operating system's behavior. Most of these require some degree of low-level knowledge about how the operating system works. The usual mode of analysis for such systems is text. If one samples the activity of a single process, they are likely to be inundated with several thousands of nearly indecipherable lines of output. Figure 1 shows a snippet of the textual output that is spewed by executing the command `sample` on a running instance of the Google Chrome application from the terminal in a Mac OS environment. There are over 1,200 lines of text associated with a 10-second sample. Most of these lines are unintelligible to those without expert knowledge of the Mac OS.

```
1    Analysis of sampling Google Chrome (pid 304) every 1 millisecond
2    Process:         Google Chrome [304]
3    Path:            /Applications/Google Chrome.app/Contents/MacOS/Google Chrome
4    Load Address:    0xc9000
5    Identifier:      com.google.Chrome
6    Version:         27.0.1453.110 (1453.110)
7    Code Type:       X86 (Native)
8    Parent Process:  launchd [262]
9
10   Date/Time:       2013-06-08 15:05:58.996 -0400
11   OS Version:      Mac OS X 10.8.4 (12E55)
12   Report Version:  7
13
14   Call graph:
15       7663 Thread_1921   DispatchQueue_1: com.apple.main-thread  (serial)
16       + 7662 ???  (in Google Chrome)  load address 0xc9000 + 0xf55  [0xc9f55]
17       + ! 7662 main  (in Google Chrome) + 24  [0xc9f78]
18       + !   7662 ChromeMain  (in Google Chrome Framework) + 41  [0xcf7f9]
19       + !     7662 ???  (in Google Chrome Framework)  load address 0xcd000 + 0x60d9d0
     [0x6da9d0]
20       + !       7662 ???  (in Google Chrome Framework)  load address 0xcd000 +
     0x60e69b [0x6db69b]

     ...

1256 0x99ee6000 - 0x99ef2ff7  com.apple.NetAuth (4.0 - 4.0)
     <4983C4B8-9D95-3C4D-897E-07743326487E>
     /System/Library/PrivateFrameworks/NetAuth.framework/Versions/A/NetAuth
```

Fig. 1. Example output from calling the command `sample` for a 10-second duration on the Google Chrome application.

Computing occurs in a real time stream of events. There is structural regularity supporting many of the tasks that are performed by a computer. Under the correct conditions, humans are adept at perceiving acoustic pitch, time, and timbre patterns [1]. Sonification is the use of non-speech audio to convey information [2]. We suggest that the sonification of the operating system is a natural mapping, one which may allow us to easily perceive computational anomalies. By exercising creative control over such a mapping, we may design an efficient means for conveying normally unobservable but important information concerning our computing environments.

This paper presents SOS, a framework for allowing users to sonify aspects of any computer running a Unix-like operating system. The architecture is simple. The source code is lightweight and utilizes only open-source packages. We provide an implementation accompanied by a couple of demos online; readers are encouraged to write synthesis and tracing scripts to meet their sonification requirements. In the following section we provide background discussion about the sonification of computer environments. We detail the architecture of SOS in Section 3. Section 4 gives concluding remarks.

2 Background

Early computer scientists often used sound as a means for monitoring the operation of their machines. Many computers built in the 1940s and 50s such as SEAC and SWAC had an amplifier and speaker attached to one or more registers of the machine. If a computer did not come equipped with an audio transducer, it was often added later—commonly on the lowest bit of the accumulator [3]. Fernando Jose Corbatò reports that the audio amplifier on one register of the Whirlwind computer allowed him to debug his programs by hearing a “signature racket” [4]. Once accustomed to the sound of the machine, the user could identify specific components of the executing program and listen for bugs or loops.

Computer operators have used AM radio to sonify their machines. Many computers, notably the IBM 1401, emitted radio frequency signal that could be picked up by an AM radio receiver when it was placed near the console or other high-speed circuits. Since computers would often take several hours to finish a task, engineers could free themselves by placing an AM radio next to an intercom. The computer operator would monitor their machine and track program termination [5]. Others have found ways to compose music from the leaking RF signal of an IBM 1401 [6].

These methods of troubleshooting computers, sometimes referred to as *sonic debugging* [7], continued well into the 1980s. Some computer musicians were inspired by the folklore of listening to endless loops and optimization problems on old mainframe computers. There is a tradition of musicians sonifying their computers for compositional purposes. For instance in 1986 computer musicians Phil Burk, Larry Polansky, and Phil Stone used an Amiga’s stack memory to modulate the frequency and amplitude of simple waveforms in the piece *mod.mania I, II, and III* [8].

More sophisticated methods of computer sonification emerged in the 1990s with integrated development environments and wider availability of personal computers. Joan Francioni and Jay Alan Jackson describe a system for the auralization of parallel program behavior. Their work gave evidence that sound was effective in depicting timing information and patterns related to the execution of a program [9]. LogoMedia is a sound-enhanced programming environment which allows users to associate non-speech audio with program events while code is being developed [10]. A similar tool named LISTEN is also intended for debugging programs [11]. CAITLIN is designed for sonifying specific control sequences (i.e. IF, THEN, WHILE, etc) as unique, customizable MIDI sequences [12].

The twenty-first century has seen increased interest in sonifying network data. Peep (The Network Aurilizer) by Michael Gilfix and Alan Couch creates a soundscape with events triggered by network state information [13]. Mark Ballora et al. have introduced a system that sonifies web logs or web traffic in real-time via Python and SuperCollider [14]. Michael Chinen has developed FuckingWebBrowser, a simple open-source web browser that converts memory state into audio [15]. For a more thorough background on the sonification of computers, see [16].

The framework we suggest uses DTrace [17], an extensive dynamic tracing library. DTrace inserts program code into the kernel and hence allows real-time, direct access to the basic operations of the operating system. We are unaware of an extant framework that links a tracing library like DTrace to acoustic events. The next section gives detail about the system, which may be used for strict data mapping or for more general compositional purposes.

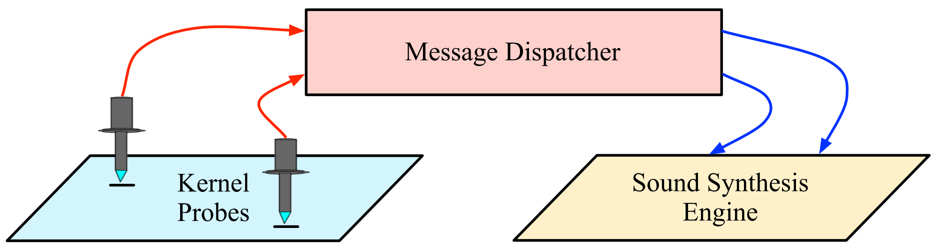

Fig. 2. General SOS architecture. Several DTrace probes are instrumented. The message dispatcher listens for probe firings and sends OSC messages to a predetermined port as necessary. The sound synthesis engine listens for messages from the message dispatcher and synthesizes acoustic events in real time.

3 Architecture

We describe the framework of our operating system sonifier, SOS. There are three modules: the message dispatcher receives messages from the kernel probes and sends messages to the sound synthesis engine (see Figure 2). It is up to the user to specify which operating system events to monitor and auditory events to generate. The SOS framework provides easy mapping for sonification. We provide detail on each module below. The source code and several demos are provided at `https://github.com/woodshop/dsonify`.

3.1 Kernel Probes

We use DTrace to monitor arbitrary system calls executed by the kernel. DTrace is a dynamic tracing framework originally developed by Sun Microsystems for the Solaris operating system and that has been widely adopted by many Unix-like operating systems, including Mac OS. DTrace allows one to modify the operating system kernel by attaching probes at various locations within the kernel execution. The set of available probes is quite large and depends on the OS.

DTrace probes are instantiated by writing a script in the D language, a subset of the C programming language. An example script is shown in Figure 3. The program prints the message "open" to the standard output every time the system call `open` is entered. The `pragma` statement on Line 2 indicates that the probe should report back at a rate of ten thousand cycles per second. These messages are passed to the message dispatcher.

```
1    #pragma D option quiet
2    #pragma D option switchrate=10000hz
3
4    syscall::open:entry
5    {
6        printf("open");
7    }
```

Fig. 3. A DTrace program. The program inserts a probe in the `open` kernel system call. When the system call is entered the probe fires and prints "open" to the standard output.

3.2 Message Dispatcher

The message dispatcher receives messages from the kernel probes and sends messages to the synthesis engine. The dispatcher is programmed using the Python language. Message passing from DTrace is handled using the Python module `python-dtrace` [18], a DTrace consumer for Python. The message dispatcher instantiates DTrace in a separate thread and listens for `printf` messages that are intercepted by `python-dtrace`.

We use the Open Sound Control (OSC) [19] protocol to send messages to the sound synthesis engine. Each time a probe message is received, an OSC message is sent along with an optional value. The sound synthesis engine receives the message and synthesizes the appropriate sound event. The python interface for OSC is handled by the pyOSC module [20].

3.3 Sound Synthesis Engine

Sound synthesis is handled by ChucK [21], a "strongly timed" engine for sound synthesis. It has been widely used for performances by laptop ensembles, live coders and multimedia artists.

```
1    OscRecv recv;
2    9000 => recv.port;
3    recv.listen();
4    recv.event( "open", "s" ) @=> OscEvent oe_open;
5
6    main();
7
8    fun void main() {
9        spork ~ inst_open();
10       1::day => now;
11   }
12
13   fun void inst_open()
14   {
15       Sitar s => Pan2 p => dac;
16       -0.75 => p.pan;
17       0.2 => s.gain;
18       while(true) {
19           oe_open => now;
20           while (oe_open.nextMsg() != 0) {
21               oe_open.getString();
22               440.0 => s.freq;
23               1.0 => s.noteOn;
24           }
25       }
26   }
```

Fig. 4. A ChucK program. The `main` function starts a thread that runs a real-time audio synthesizer.

Upon executing the message dispatcher, a ChucK engine is instantiated and the synthesis script provided by the user is loaded into memory. The synthesis engine listens for OSC messages over an arbitrary port. Figure 4 shows a short ChucK program. Line 2 instructs ChucK to listen for incoming OSC messages on port 9000. The synthesizer designated in `inst_open` is executed in a separate thread (the ChucK terminology for initiating a new thread is `spork`). The message dispatcher sends the OSC message "open" accompanied by a string. The string may be employed as a control value for the synthesizer. (The example shown in Figure 4 does not use the string.)

The ChucK programming language is simple yet allows one to mix any combination of synthesizer parameters with incoming OSC messages. Hence the user may sonify their operating system in a manner that conveys important information about OS events. Alternatively, the user may use OS data as a rich resource for music composition or performance. The interaction between OS sonification and the additional computational resources demanded by audio synthesizers suggests an area of musical feedback worthy of exploration.

4 Conclusions

In this paper we presented a system for sonifying Unix-like operating systems. The user must supply a DTrace and ChucK script to SOS specifying sonification mappings. DTrace is used to monitor the actions of the operating system kernel. For each action, a message is passed to a message dispatcher, which subsequently sends an OSC formatted message to a ChucK synthesis engine. Given the large quantity of system calls per second, sonification allows users to observe dense information in a mode more natural than text. By sonifying the operating system, a user's visual attention is freed to focus on other details.

The code along with two demos for SOS can be found on github at `https://github.com/woodshop/dsonify`. In the first demo integral file management operations such as file opens and file writes are sonified. The demo gives the user an idea of how much file access is occurring in the background. Our second demo allows the user to monitor all outgoing Internet Protocol version 4 (IPv4) connections, along with the port number. Such sonification allows a user to be alerted when non-standard ports are being utilized. This information may be helpful to discover when applications are contacting the outside world without the user's knowledge. See the online documentation for more information concerning the auditory events used in the demos. The online accompaniment to [22] (from which we have based our demos), found at www.dtracebook.com, is an excellent primer for DTrace programming.

There have been demonstrations in the literature and online for sonifying specific aspects of the computing environment. To the best of our knowledge, SOS is a first attempt at providing a general framework that allows sound designers to link unobserved computing events to their acoustic surroundings. Given the high flexibility of ChucK and the enormous power of DTrace, we believe that SOS provides ample opportunity for growth and exploration of OS sonification or music composition using the OS as an agent. We encourage readers to download the code and contribute additional demos to the community.

References

1. Bregman, A.S.: Auditory scene analysis: the perceptual organization of sound. MIT Press, Cambridge, Mass. (1990)

2. Kramer, G.: Auditory display: sonification, audification, and auditory interfaces. Proceedings volume 18, Santa Fe Institute studies in the sciences of complexity. Addison-Wesley, Reading, Mass. (1994)
3. Thelen, E.: 1401 music, sounds, and movies. Online comment by Michael Mahon: `http://ibm-1401.info/Movies-n-Sounds.html`
4. Frenkel, K.A.: An interview with fernando jose corbato. Commun. ACM **34**(9) (September 1991) 82–90 Interviewee-Corbató, Fernando Jose.
5. Johnson, L., Hardy, A.: Oral history of LaRoy Tymes. Online: `http://archive.computerhistory.org/resources/access/text/Oral_History/102657988.05.01.acc.pdf` (June 2004)
6. Andrews, R.: IBM 1401 mainframe, the musical. Online: `http://www.wired.com/culture/art/news/2007/07/IBM1401_Musical` (2007)
7. Personal email corresondence with composer Larry Polansky. (June 2013)
8. Burk, P., Polansky, L., Stone, P.: mod.mania I, II, and III. Composition notes online: `http://music.dartmouth.edu/~larry/misc_writings/program_notes/spareparts/Page2_BW.png` (1986)
9. Francioni, J.M., Jackson, J.A.: Breaking the silence: auralization of parallel program behavior. J. Parallel Distrib. Comput. **18**(2) (June 1993) 181–194
10. DiGiano, C.J., Baecker, R.M.: Program auralization: sound enhancements to the programming environment. In: Proceedings of the Conference on Graphics Interface '92, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1992) 44–52
11. Boardman, D., Greene, G., Khandelwal, V., Mathur, A.: LISTEN: a tool to investigate the use of sound for the analysis of program behavior. In: Computer Software and Applications Conference, 1995. COMPSAC 95. Proceedings., Nineteenth Annual International. (1995) 184–189
12. Vickers, P., Alty, J.L.: Siren songs and swan songs debugging with music. Commun. ACM **46**(7) (July 2003) 86–93
13. Gilfix, M., Couch, A.L.: Peep (the network auralizer): Monitoring your network with sound. In: Proceedings of the 14th USENIX conference on System administration. LISA '00, Berkeley, CA, USA, USENIX Association (2000) 109–118
14. Ballora, M., Giacobe, N.A., Hall, D.L.: Songs of cyberspace: an update on sonifications of network traffic to support situational awareness. In: Proc. SPIE. Volume 8064. (2011)
15. Chinen, M.: Fuckingwebbrowser. Online: `http://michaelchinen.com/fuckingwebbrowser/` (2010)
16. Hermann, T., Hunt, A.: The sonification handbook. Logos Verlag, Berlin (2011)
17. Sun Microsystems, Inc.: Solaris Dynamic Tracing Guide. Sun Microsystems, Inc., Santa Clara, CA (2008) Online: `http://docs.oracle.com/cd/E19253-01/817-6223/`.
18. Metsch, T.: python-dtrace. Online: `http://tmetsch.github.io/python-dtrace/` (October, 2011)
19. Wright, M.: Open sound control: an enabling technology for musical networking. Organised Sound **10** (12 2005) 193–200
20. v2lab: pyOSC. Online: `https://trac.v2.nl/wiki/pyOSC` (2013)
21. Wang, G., Cook, P.: ChucK: Strongly-timed, Concurrent, and On-the-fly Audio Programming Language. Online: `http://chuck.cs.princeton.edu/` (2012)
22. Gregg, B., Mauro, J.: DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD. Prentice Hall (2011)