Peep (The Network Auralizer): Monitoring Your Network With Sound

Michael Gilfix & Prof. Alva Couch – Tufts University

ABSTRACT

Activities in complex networks are often both too important to ignore and too tedious to watch. We created a network monitoring system, Peep, that replaces visual monitoring with a sonic 'ecology' of natural sounds, where each kind of sound represents a specific kind of network event. This system combines network state information from multiple data sources, by mixing audio signals into a single audio stream in real time. Using Peep, one can easily detect common network problems such as high load, excessive traffic, and email spam, by comparing sounds being played with those of a normally functioning network. This allows the system administrator to concentrate on more important things while monitoring the network via peripheral hearing.

This work was supported in part by a USENIX student software project grant.

Introduction

Are your systems and network functioning correctly? Can you be sure at this moment? Every administrator has some need to be able to answer these or similar questions on an ongoing basis.

Current approaches to live monitoring of network behavior (such as Swatch [10], mon [4], and their many relatives) can send email or page responsible people when things seem to go wrong. These tools are both visual and intrusive; operators must either be interrupted by alerts or periodically suspend other work to check on network status. Furthermore, these approaches are highly *problem-centered* and provide mainly *negative reinforcement*; the monitor notifies an operator only when problems occur. It does not, as a rule, regularly inform one when things are going well.

We created a tool Peep that represents the operational state of a system or network with a *sonic environment*. The flavor, texture, and frequency of sounds played are used to represent both proper and improper network performances, while the 'feel' of the sounds provides the listener with an approximation of network state. This environment plays in the background while the operator continues other tasks. Without looking anywhere and without interrupting other pressing activities, the operator can hear *peripherally* whether action is required.

Auralization

The idea of auralizing network behavior by playing network sounds is not new. Joan Francioni and Mark Brown [3, 5] represented parallel computer performance using a synthesizer driven by a MIDI interface. The strength of this approach, however, was also its main limitation. For music to remain pleasant, one must limit one's representations to a limited number of relatively pleasing harmonic combinations. This greatly limits what one can represent with this technique. *Earcons* [2] are the sonic equivalent of icons; sounds that are naturally associated with particular events. For example, most people associate a car horn with impatience or alert and a doorbell with someone entering a house.

Both of these approaches define the meanings of specific sounds or particular combinations in isolation. Combining sounds is difficult unless they are consonant either musically or environmentally, that is, that the sounds naturally occur together and 'sound right' in combination. Natural sounds have an advantage over music; they sound normal and pleasing in almost any combination similar to that of nature. For example, birds and frogs in wetlands can sing with virtually no coordination, and the result is still pleasing.

The Psychology of Audio Notification

What makes Peep possible is that events in networks have easily recognized natural sound counterparts. Moreover, numerous natural sounds can be played in combination while the result stays pleasing to the ear. If each sound represents some part of network function, and all are played together, the result is a *sonic ecology* in which the current state of the network can be determined moment by moment.

Peep exploits human instinct: our ability to notice a deviation from the norm with little effort, to determine what sounds right, and to discern singular important sounds from a collection of many sounds. We do these tasks with little or no conscious effort. Since computer interfaces mainly require the visual senses (and some motor skills), the audio senses are left available to perform this unconscious processing.

Furthermore, Peep takes advantage of our ability to do abstract processing. Instead of attempting the difficult and sensitive problem of determining when a network crisis has occurred or is about to occur, Peep provides contextual, continuous sound information and leaves interpretation to the listener. Decisions are based not only on the quantitative measure of things, but the relative amount and absence of things. A musician friend has often expressed to me his philosophy: "Anybody can play drums, but the great drummer concentrates as much on the feel of the notes as on the space, or absence of sound, between them." Similarly, information that is lacking from Peep's sound ambiance is just as important as the amount of information conferred and the relative magnitude is left to the judgement of the listener.

Representational Techniques

Sound representation in Peep is divided into three basic categories: *Events* in networks are things that occur once, naturally represented by a single peep or chirp. Network *states* represent ongoing events by changing the type, volume, or stereo position of an ongoing background sound while *heartbeats* represent the existence or frequency of occurrence of an ongoing network state by playing a sound at varying intervals, such as by changing the frequency of cricket chirps.

Peep represents discrete events by playing a single natural sound every time the event occurs, such as a bird chirp or a woodpecker's peck. The sounds we chose are short and staccato in nature and easily distinguishable by the listener. Additionally, we noted that certain events tend to occur together and found it convenient to assign them complementary sounds. While monitoring incoming and outgoing email on our network, we noticed that the two events were often grouped together, since both types of email were usually transferred in a single session between mail servers. To better represent this coupling between incoming and outgoing email events and make the representation sound more natural, we used the sounds of two conversing birds. Thus, a flood of incoming and outgoing email sounds like a sequence of call and response, making the sound 'imagery' both more faithful to our network's behavior, as well as more pleasing to the ear.

State sounds correspond to measurements or weights describing the magnitude of something, such as the load average or the number of users on a given machine. Unlike events, which are only played when Peep is notified of them, Peep plays state information constantly and need only be signaled when state sounds should change. Peep represents a state with a continuous stream of background sounds, like a waterfall or wind. Each state is internally identified as a single number measurement, scaled to vary from extremely quiet to loud and obnoxious. Background sounds should be soothing while the network is functioning normally. However, when the administrator is annoyed, he will know that action is required.

Heartbeats are sounds that occur at constant intervals, analogous to crickets chirping at night. A common folk tale is that one can tell the temperature from the frequency of cricket chirps; likewise we can represent network load as a similar function. Intermittent chirps might mean low load, while a chorus might mean high load. Heartbeats can also report results of an intermittent check (or ping) to see if a given machine, device, or server is functioning properly.

Humans are very apt at recognizing when continual background sounds change, making problem detection swift and simple. If your email server dies, chances are that you will not receive any email warning of the problem. But the crickets will have stopped chirping. The heartbeats provide an effective method for monitoring the functionality of your network and being alerted of a problem when all else fails, through the absence of sound. Likewise, the administrator need not fear about monitoring his Peep server; if it dies, he will be immersed in sudden silence!

Sound representation depends very much on personal taste. Peep aims to provide users with a choice of themes such as *wetlands* (the current theme available) or *jungle*. Within a theme, sounds are classified according to the network events they most appropriately express. Although the two chorusing birds were used to represent incoming and outgoing mail in the previous example, the two bird sounds could have been used for any type of coupled event behavior. These classifications help the user make decisions on what sounds to use from his collection of favorites.

We also recognize that distinguishing sounds can be difficult if, for example, several similar bird sounds are used in a single theme. As the theme repository provided with Peep expands, we hope it will address a wide range of network situations and personal tastes.

Scalability and Flexibility

The Peep architecture was designed to be versatile and scalable. The architecture is based upon a producer/consumer relationship between distributed monitoring processes that watch the network and servers that actually play sounds. Producers alert consumers to events and state changes via short UDP messages, as shown in Figure 1.

This architecture allows the receipt of status reports from any number of devices or nodes. Producers (the monitors in Figure 1) monitor network behavior and report events and states while consumers take their input from the producers and play the appropriate sounds. Producers can be pointed at several sound generators simultaneously, e.g., a lab full of Linux workstations, for a truly immersive experience!

Producers are executed as daemons on machines with access to information sources. This eliminates the need to send copious amounts of sensitive log or machine information across the network to a centralized monitoring server. The packets sent to the consumer contain only sound representation information and would be of little use to a snooper without access to the Peep configuration file.

The Peep system was designed to take advantage of existing system administration tools. Server and client configuration information is stored in the same configuration file. This allows centralized control of Peep via simple file distribution via NFS or other widely accepted mechanisms such as CFEngine [6, 7, 8] and rdist [9].

Clients provided with the Peep distribution are 'lightweight' Perl scripts. Each client functions strictly within one problem domain: it addresses its original intended purpose and no more. This keeps client code simple, easy to debug, and easy to customize.

We also wanted clients to run in the background and utilize as little resources as possible. Our log probing client, LogParser, watches log files and uses regular expressions to determine when particular events have occurred. Because of the way regular expressions are mapped in memory, scanning a single log for many different text patterns can become memory intensive. Instead, we designed LogParser to distribute monitoring overhead. Multiple instances of LogParser can run on separate feeds around the network, each instance searching for only a few textual patterns in the local system logs. This allows the system administrator to take advantage of the distributed computing power of his network, rather than waste what is often an abundance of idle resources in the hands of naive users. Peep aims to provide administrators with several means of implementing monitoring. Administrators still have the option of directing all log entries to a single machine should they so desire, at the cost of increased network bandwidth. Furthermore, the distributed method can be combined with the single-machine method with no effort on the administrator's part.

Expanding the capabilities of Peep to fit your own needs is simple. Perl libraries handle all the low-level details, so writing scripts for event, state, and heartbeat-driven feeds can be quick and painless. LogParser can also be easily configured to scan a log for new events via additional regular expressions.

The Peep Protocol

Peep was designed to allow centralized management of its distributed architecture. The Peep protocol uses auto-discovery to dynamically bind clients and servers together upon startup. Peep configuration also uses a class mechanism to define groups of clients that should all report data to the same servers.

Peep was originally designed to use TCP for communication between clients and servers but communication over UDP proved much more efficient and effective. The main strength of TCP is its reliability. However, this reliability comes at the cost of greater bandwidth usage. Extra packets must be sent to ensure that transmissions were received correctly and in the proper order. Peep does not require packets to be ordered in any way – nor for packet transmissions to be reliable – since the representation of the state of the network is an approximation rather than a precise depiction. In any case, the human ear has no way of distinguishing the exact order of events when events rapidly arrive at the Peep server; indeed, the resulting sounds seem simultaneous.

The statelessness of UDP provided another benefit: clients and servers can be stopped and restarted

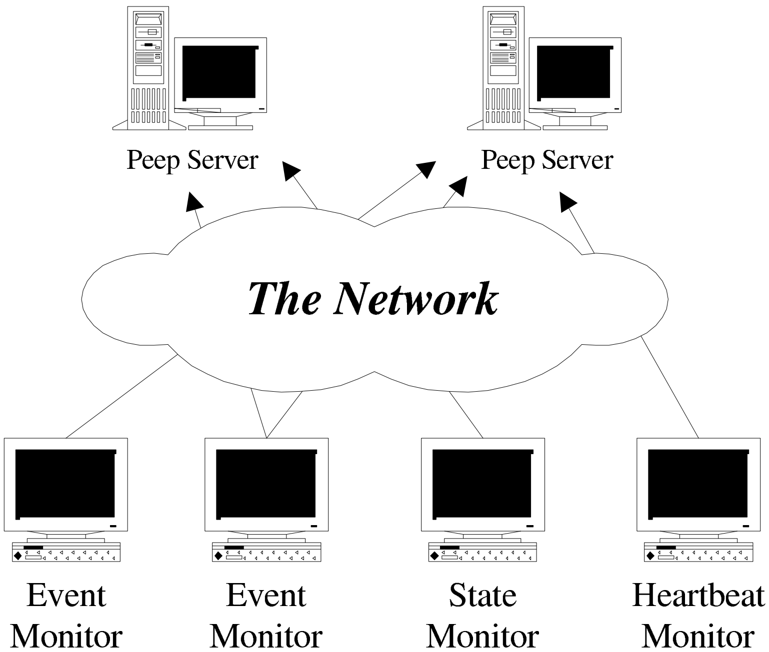

Figure 1: The Peep architecture.

without affecting one another. We wanted users to be able to write their own clients with minimal hassle. Avoiding connection management keeps clients simple and allows one to readily write Peep clients without making use of the included Perl libraries.

One drawback to using UDP is that clients have difficulty determining when servers crash. If this problem is not addressed, a client will continue to provide data to a non-existent server forever. Peep deals with this problem by combining a leasing mechanism with auto-discovery. This combination provides safe, dynamic, real-time bindings between clients and servers.

Peep's auto-discovery mechanism uses a domain-class concept to maintain bindings between clients and their respective servers. When a server initializes, it broadcasts its existence to the subnets associated with its classes and announces the classes of which it is a part. The clients that are members of those classes register themselves with the server and begin sending it packets. Conversely, should a client start up and broadcast its existence, the servers associated with its class will tell it to begin sending. A broadcast only occurs once during the initialization of each client or server, after which a list of hosts is maintained on both sides and communications are direct. Both clients and servers can belong to multiple classes at the same time and clients can communicate with many servers concurrently.

Leasing is used to ensure that clients do not waste network bandwidth and system resources sending packets to servers that are no longer listening. The server sends a lease time to the client during auto-discovery. Just before the lease expires, the server tells the client to renew the lease. The client responds by telling the server that it is still alive and still needs to know about lease information. If the client has not heard from a server after the lease time has expired, it will no longer send packets to that server. Similarly, if a server does not receive lease acknowledgement from a client, it will no longer attempt to renew its lease with that client.

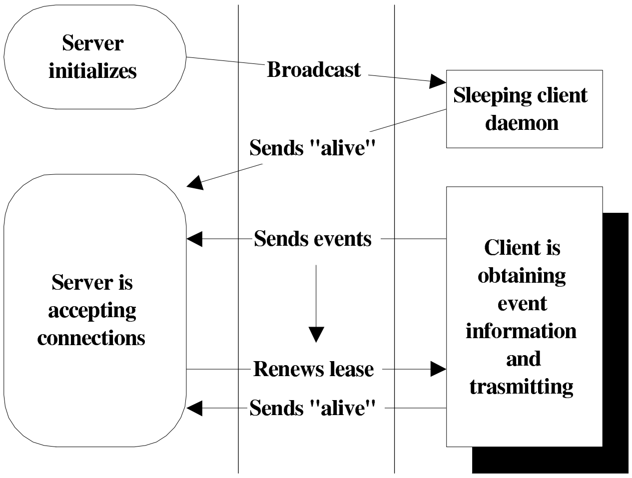

Figure 2: A server initialization.

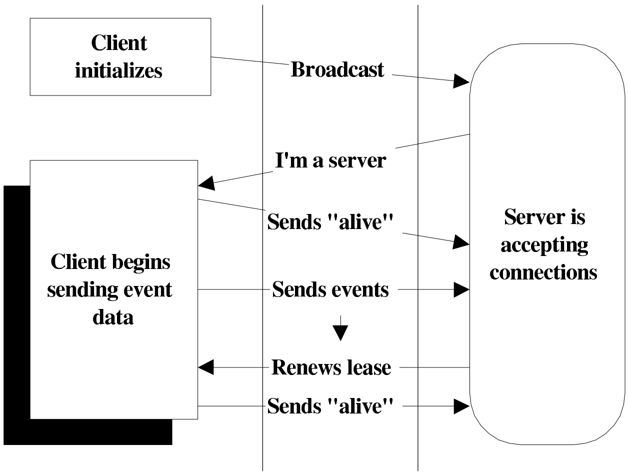

Figure 3: A client initialization.

The auto-discovery and lease mechanisms greatly ease the burden on the system administrator. The system administrator can then use a file distribution mechanism, like CFEngine, to add client and server daemons to a machine's background processes. Clients will sleep until a server becomes available, and will send packets only while that server stays available.

Alternatively, system administrators may decide to dedicate a machine to run Peep software and want all clients to execute on a single machine. In this situation, broadcasting becomes totally unnecessary and inefficient. Instead, the user can disable the auto-discovery mechanism. Clients will then become dumb clients, continually processing and sending event information to a server throughout the course of their lifetimes. Peep also provides the user the choice of mixing and matching, applying distributed and centralized configurations where they make sense.

In terms of robustness, the Peep protocol has version identification, room for future expansion, and type identification. Upgrades should allow older clients to work with newer servers and vice versa. Communications are done using one-byte quantities to represent attributes, and strings for anything more complex. This allows us to avoid any external data representation issues, making the protocol more portable.

Details of this protocol are hidden inside a Perl client interface library provided with Peep. The Peep library demands little expertise. To create a client with all of the library's benefits, programmers need only initialize the library with their application name and tell the library what information to send. Initializing the library parses the Peep master configuration file, so programmers need not do it themselves. This allows client design to be as simple or as complicated as the user desires. We hope that the simplicity of writing clients with the Perl library will encourage users to write their own client applications and share their code with others.

Configuring the Peep System

How one configures Peep is very much dependent on whether you choose to use single or multiple nodes. The generalized Peep installation is a four-step process: downloading the source and a sound package, compiling the server, editing the configuration file, and deploying clients.

The Peep server package uses the gnu autoconf package to make configuration and compilation easy. Support for tcp_wrappers [11] can be added as an option. Peep comes with two generic sound modules. One handles generic /dev/audio support while the other takes advantage of ALSA [1] on Linux systems. The configure package will default to ALSA drivers over generic support, if present. Special support for the Sun audio jack is also provided.

After compilation, the next step is to tell Peep which sounds to associate with which events, the classes to which your clients and servers belong, and your client configurations. A simple Peep configuration file is shown in Figure 4.

```
class myclass
  broadcast 130.64.23.255:2000
  server swami:2001
end myclass

client LogParser
  class myclass
  port 2000
  config
   #Name|OptLetter|Location|Priority|RegX
   out-mail    0   1  "sendmail.*:.*from"
   inc-mail    I  255 0  "sendmail.*:.*to"
  end config
end client LogParser

events
  #Event Type|Path|# sounds to load
  out-mail /path/sounds/peep1a.*  1
  inc-mail /path/sounds/peep2a.*  1
end events

states
  #Event Type|Path|# sounds| Fade time
  loadavg /path/sounds/water.*  5  0.3
end states
```

Figure 4: An example peep.conf.

Class definitions consist of two lines: one specifying broadcast zones and another specifying which servers are part of that class. Several broadcast zones and servers can be specified. Clients and servers can be part of several classes and will broadcast all the classes to which they belong during initialization. Putting multiple servers in a class (or making a client a member of multiple classes) is an easy way to have a single client dump data to multiple servers.

The 'events' and 'states' sections tell Peep servers to associate a name with a group of sounds. Filename descriptions in the Peep configuration file have a trailing asterisk extension followed by the number of sounds to load. Peep expands each asterisk into a two-digit number and loads, in ascending order, the number of sounds specified. All of the sound files loaded for a single entry then correspond to a single event. Every time that event occurs, the server will randomly play one of the associated sounds. This randomness makes the sound ambiance more natural. Heartbeats are created from streams of normal events from a client at suitable intervals. For state sounds, the server randomly strings together sound segments to create a non-repeating, random-sounding background ambiance. To keep transitions between sound segments sounding natural, the user can specify a linear fade time between segments.

The final step is to configure and deploy some of the clients provided with Peep. Two of those are discussed here: Peck and LogParser.

Peck

Peck is a command-line utility provided with Peep. It allows the user to tell a server to play (and how to play) a given sound. Peck is an example of a dumb client and bypasses the auto-discovery and leasing mechanisms. Event and state attributes are specified on the command-line and delivered directly to the server. Some command-line options apply to event sounds and others to background sounds, but the user need only remember a small number of options to get the Peep server to play some interesting things. Peck can be called with appropriate arguments from a shell script if a user does not wish to use a client library. Ideally, one should only utilize Peck to talk to servers on the same physical machine, or to report very infrequent events since Peck's inability to use auto-discovery and leasing capabilities means that calling applications will have no knowledge of the state of the receiving server. Peck is handy for a variety of simple tasks, including debugging installations, testing how things sound together, experimenting with Peep's capabilities, and interfacing Peep with other monitoring systems (such as an existing Swatch or mon installation).

LogParser

A simple log analyzer, similar to Swatch, is also provided with Peep. LogParser takes advantage of Peep's auto-discovery and leasing mechanisms. It is also an efficient distributed tool. LogParser reads its entire configuration but only searches for and remembers textual patterns specified on the command-line. It was designed to have multiple instances run on several different machines, each scanning for different sets of textual patterns on each client machine.

LogParser is flexible, easy to configure, and provides a simple way to access Peep's capabilities for representing events and states. It analyzes log messages as they are added to the log file and scans them for regular expressions. LogParser uses simple configuration syntax to generate command-line options and determine which sounds to associate with which particular events. Several options follow:

- The **priority** of the event ensures that no matter how many network events hit the Peep server, the most important ones will be played first and foremost.
- The **stereo location** of the event, aside from pleasing the true audiophile, helps the user distinguish and even locate an event. Sonic locations can even be assigned to correspond to the actual locations of machines on the network. Future versions of Peep might include a visual sound location map to exploit this.
- A **regular expression** that tells LogParser how to find the event in a log file. Users with experience with Awk/Perl pattern matching will appreciate this feature while others may find writing these difficult. We feel this is the easiest way to extend the capabilities of Peep without doing any sort of programming.

Directives in the LogParser configuration can be enabled or disabled via command-line options. Each line of the LogParser configuration corresponds to a user-specified single-letter option. In Figure 4, incoming and outgoing mail are mapped to command-line options "I" and "O", respectively. Thus, an invocation of LogParser searching for incoming mail might look as follows:

```
LogParser -events=I
        -logfile=/var/log/messages
```

Should the user forget the options, a help option will conveniently generate a list of user-configured options.

A single instance of LogParser can scan numerous logs simultaneously. It can send event streams to multiple servers automatically via the auto-discovery and domain-class mechanisms. These features provide the user with a myriad of options for structuring the architecture of Peep within a network.

Peep Performance under Pressure

To deal with copious amounts of incoming network data, Peep has a queuing and windowing system that handles large numbers of simultaneous events. This ensures that events are played in the order of receipt and in accordance with their particular priority. Peep will also discard events from its queue if too much time elapses between receipt and playtime, in order to keep events relevant.

Peep plays sounds by mixing sources in software. Since having large numbers of simultaneous voices can become computationally expensive, the user can tweak Peep's performance by changing the number of voices used when mixing sound. Less mixing voices tend to mean that the Peep's queuing and windowing system gets more usage, but the two always strike a balance to keep events accurately positioned in terms of time of occurrence.

It is difficult to send events to a Peep server fast enough to fill a queue on a Pentium II 400 and during testing, this required the use of an infinite loop. If the Peep server does manage to become overloaded, it only falls behind time-wise, adding a delay between the real-time event and the playing of its counterpart. Peep will preserve the general order and users will still be able to diagnose problems based upon the relative frequency of events. The delay experienced only applies to events and heartbeats; state changes occur instantaneously. In a worst case scenario, should the queue manage to fill up while new events are still arriving, Peep will begin discarding the oldest events from the queue, attempting to give the best approximation of network activity.

A Brief Overview of Implementation

The inner-workings of a Peep server are based upon the interactions between three execution threads as shown in Figure 5: the listener, the engine, and the mixer. The listener handles all communications with the client, discovering clients via auto-discovery and keeping track of client leases. Upon receipt of event or state data, the listener thread places the information into a queue to be processed by the engine. The engine works closely in conjunction with the mixer to keep track of the priority of incoming and currently playing sounds. The engine also tries to find the best available mixing channel on which to play the incoming events and informs the mixer of the necessary parameters to properly represent the information. Should a suitable mixing channel not be found, the engine will place the events into a priority queue, ensuring that the mixer will play the most important events as soon as mixing channels free up. The mixer performs the processing necessary to produce Peep's output. This process involves scaling each sound's volume, as well as fading between state sounds. The mixer must also check the engine's event queue and ensure that queued, older events have priority as soon as mixing channels free up.

Critique

From our perspective, the design of Peep is very robust and portable. We decided, however, that support for generic audio hardware was more important than efficiency of memory and processor usage on the server side. Peep utilizes Linux ALSA and OSS drivers, as well as the Solaris /dev/audio interface, to avoid device incompatibilities. This is done at the expense of ignoring commonly available device-dependent hardware-based mixing in favor of mixing in software. Software mixing did afford us one advantage that hardware cannot guarantee: users will always get the benefit of sound processing incorporated into Peep regardless of the hardware. Future plans do include support for hardware-based mixing on a selected number of audio cards.

An invisible limitation of Peep is that creating accurate natural venues of consonant sounds is both an art and very labor-intensive. Due to copyright limitations on existing natural sound collections, Prof. Couch has spent many hours with a Telinga parabolic nature microphone and Sony DAT or digital minidisc recorder in search of the perfect bird. Sounds we collected required significant post-processing, including high and low-pass filtering and noise reduction, before they were free of enough normal background noises to serve as event sounds. Collecting state sounds proved even more difficult, with the sound of wind being the most difficult. The challenge was to collect 'desirable noise' without impurities such as car horns and airplane engines.

In spite of the excellent guidance on the recording of natural sounds that we obtained from the Cornell Ornithology website [13], the Stokes Field Guide to Bird Songs [14, 15], and the British Library National Sound Archive [12] we are not ornithologists and apologize in advance for any gross mislabeling of sounds included with Peep! Nonetheless, we have made significant progress in providing a Wetlands venue, and are planning others in the future.

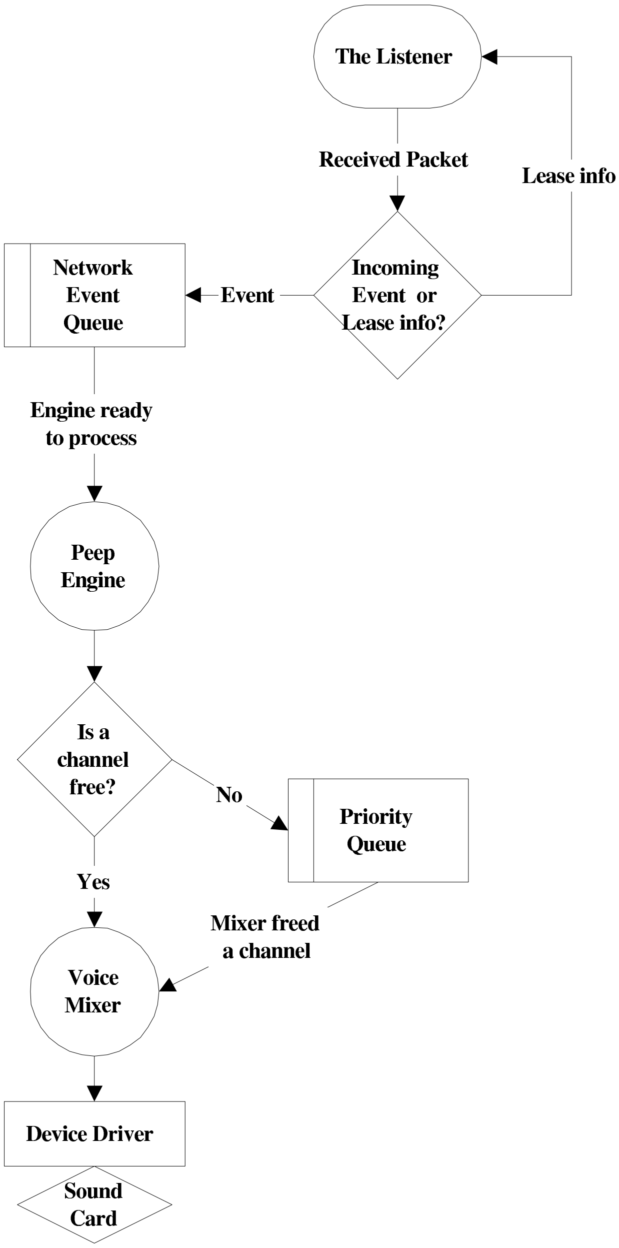

Figure 5: The Peep server's internal structure.

Configuring a Peep theme pleasingly can be non-trivial, especially when choosing which sounds should be associated with which events. The process of choosing sounds can often be a very lengthy. Since sounds chosen vary according to personal taste and the situation they are attempting to describe, we hope to provide several different preset configurations for our users after the tool has had more exposure.

Peep is relatively young and prior to this publication has received very little public usage. We hope we have anticipated and met the needs of a wide range of network implementations. However, only public usage and time will tell.

Future Work

We want to see several other capabilities added to Peep servers to better represent network events. One idea is 'log dithering'. Due to block buffering, many log files are updated in erratic bursts so that several events are written to the log file and reported by LogParser as simultaneous. A dither time would space out how the events are played so they have a truer representation.

We also want to represent state sounds in a way that better models the way the human ear works. Since the ear hears amplitudes on an exponential scale (in dB), we want to scale state measurements exponentially so that they better approximate what the human ear considers truly loud. This still may not satisfy our vision of having a storm break loose when a machine is overloaded.

We may also allow sounds to change in nature with volume. A small stream might become a river rapid when a state measurement, such as load average, increases. State sounds might be represented by three or four different collections of sounds to achieve a 'thunderous' effect. A final item on the server wishlist is pitch bending: the ability to play sounds at different frequencies. Using this capability we could generate birdcalls at different pitches and then combine them together to create the effect of a chorus of distinct birds from a single sample.

We would also like to add a GUI to ease the process of configuring sounds for Peep. Since we plan on having several different sound classifications, a sound browser would be a welcome addition. The interface would let the user play several sounds simultaneously so they could get a feel for how things would sound in various situations. This will most likely be the next major addition to the Peep software package.

Lastly, we hope a few brave users will contribute homegrown scripts and configurations to the project so that we can establish an archive and ease the process of making a new installation.

Conclusions

This work began two years ago by trying to define what constitutes 'normal' behavior of a network and how to take action to rectify 'abnormal' behavior. This proved infeasible because normalcy depends as much upon policy decisions as upon many pre-existing conditions. These conditions exhibit complexities and intricacies that are difficult to depict via traditional methods.

Our sound ecology depicts normalcy in a new way. Things are normal when Peep "sounds like it did yesterday," regardless of the intricacy of the depiction. Our innate human abilities to detect these differences are more acute than one may realize. When things sound different, we may not know why, but we can tell that *something* has changed.

Traditional tools look for specific problems while Peep only tells the listener about potential problems. In that respect, Peep will outlast traditional problem-detection tools because it portrays the general problem and no more. And unlike other tools, Peep is non-intrusive. One doesn't need to pay much attention to Peep in order to benefit. We don't want you to. We just want you to sit back, and listen.

Availability

The current revision of Peep is 0.3.0alpha and is is currently freely available from http://www.eecs.tufts.edu/peep/. A demo of Peep's capabilities will also be provided on the website in .wav format so users can know what they're getting into before they install it.

Acknowledgements

Thanks to USENIX for funding this project and making it possible. Additional thanks goes to Andy Davidoff for contributing many great design ideas throughout the course of Peep's development and for being one of the first to embrace Peep software.

Biography

Michael Gilfix was born in Winnipeg, Canada and presently resides in Montreal, Canada where he attended high school at Lower Canada College. He is currently a junior at Tufts University, where he is completing his undergraduate degree in electrical engineering and his masters in computer science. His interests include guitars, music, and computers in all ways, shapes, and forms. While completing his degrees, he is currently practicing the art of system administration in Tufts' Electrical Engineering and Computer Science department. He will be graduating in 2003. He can be reached via electronic mail as mgilfix@eecs.tufts.edu . Reach him telephonically at +1 617-627-2804.

Alva L. Couch was born in Winston-Salem, North Carolina where he attended the North Carolina School of the Arts as a high school major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M.I.T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School. Returning to school, he received an M.S. in Mathematics from Tufts in 1987, and a Ph.D. in Mathematics from Tufts in 1988. He became a member of the faculty of Tufts Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Electrical Engineering and Computer Science at Tufts. He can be reached by surface mail at the Department of Electrical Engineering and Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He can be reached via electronic mail as couch@eecs.tufts.edu . His work phone is +1 617-627-3674.

References

[1] Advanced Linux Sound Architecture, http://www.alsa-project.org .

[2] G. Kramer, Ed, *Auditory Display: Sonification, Audification, and Auditory Interfaces*, Addison-Wesley, Inc. 1994.

[3] J. Francioni and J. A. Jackson, "Breaking the Silence: Auralization of Parallel Program Behavior," *Journal of Parallel and Distributed Computing*, June 1993.

[4] J. Trocki, "Mon, the Server Monitoring Daemon," http://www.kernel.org/software/mon .

[5] M. Brown, "An Introduction to Zeus: Audiovisualization of Some Elementary Sorting Algorithms," *CHI '92 proceedings*, Addison-Wesley, Inc. 1992.

[6] M. Burgess, "A Site Configuration Engine," *Computing Systems*, 1995.

[7] M. Burgess, "A Distributed Resource Administration Using Cfengine," *Software: Practice and Experience*, 1997.

[8] M. Burgess, "Computer Immunology," *Proceedings LISA XII*, Usenix Assoc., 1998.

[9] M. Cooper, "Overhauling Rdist for the '90's," *Proceedings LISA VI*, Usenix Assoc., 1992.

[10] S. Hansen and T. Atkins, "Centralized System Monitoring With Swatch," *Proceedings LISA VII*, Usenix Assoc., 1993.

[11] W. Venema, "TCP WRAPPER, network monitoring, access control, and booby traps," *UNIX Security Symposium III*, September 1992.

[12] "The British Library National Sound Archive," http://www.bl.uk/collections/sound-archive, The British Library, 2000.

[13] "The Library of Natural Sounds," http://birds.cornell.edu/lns/, Cornell Lab of Ornithology, 2000.

[14] D. Stokes, L. Stokes, and L. Elliot, *Stokes Field Guide to Bird Songs: Eastern Region* (three audio CD's), Warner Books, Inc., 1997.

[15] Peterson Field Guides, *Eastern/Central Bird Songs* (three audio CD's), Houghton-Mifflin, Inc., 1999.