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

Michael Gilfix & Prof. Alva Couch - Tufts University

This page is legacy content.

Check out the current usenix

Abstract

Activities in complex networks are often both too important to ignore and too tedious to watch. We extend a network monitoring system, Peep, that produces visual monitoring with sonic 'outrigger' of sonical sounds, where each sound is a function of network traffic. Our system uses a rule-based approach to combine network state information from multiple data sources, to trigger 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 computing 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 is dedicated 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 SnmpB [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 interactive; 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 little *context*. *Context* is the information surrounding a problem, such as what else is going on at the time, or a rule, regularly affirm not 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 functioning and problems. Peep's ability to "mix" sounds from a rule-base allows it to express complex situations such as a problem happening in the background while the operator is doing other work. Without looking anywhere and without interrupting other pressing activities, the operator can hear *peripherally* whether action is required.

Auralization

The idea of sonifying network behavior by playing network sounds is not new. Joan Francioni and Mark Brown [3, 5] proposed 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 sounds and combinations of sounds. In contrast, our approach uses raw samples as its sonic primitives. Gaver [2] set the stage for this approach with his sonic work (SEDs), but audio sources with perceptual meaning, for example, most people associate a car horn with impatience or alert and a doorbell with someone entering a house.

Peep uses these sonic primitives to express the status of specific sources of information. Combinations of these primitive 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 nature 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 monitoring by ear.

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. Peep's ability to "mix" sounds from various sources allows it to express complex situations. Since computer networks are complex, and have several interrelated subsystems, we believe that Peep can perform this monitoring more pleasantly.

Furthermore, Peep takes advantage of our ability to do abstract processing. Instead of interpreting the difficult and sensitive problem of determining when a network crisis has occurred or is about to occur, Peep allows the operator to process raw sound information and leave interpretation to the human. 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, between the notes." Similarly, information that is lacking from Peep's sound ambiance is just as important as the amount of information combined 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 processes that last for a period of time and require a sound that can be looped to represent the ongoing process. Finally, *continuous* sounds are used to represent a process that is always occurring and whose frequency or speed changes to reflect the rate of change.

Peep represents discrete events by playing a single natural sound every time the event occurs, such as a third chirp or a woodpecker's peck. The sounds we chose are short and staccato in nature and have a distinct beginning and end. Conversely, we used sounds that contain no natural end or beginning to represent state because 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 hear these two happenings between incoming and outgoing email events and make the operational state of the network sound as well, we selected the sound of two frogs croaking back. Thus, a flock 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.

Finally, we represented continuous processes by modulating the magnitude of a given sound to represent the value of the process. For example, in our system, CPU loads events, which are only played when Peep is notified of them, Peep plays state information constantly and need only be signaled when sounds should change. Peep represents a state with a continuous stream of sound samples, and uses a sound filter to represent the current load. Each state is constantly identified as a single number measurement, scaled to vary from extremely quiet to loud and obnoxious. Background sounds should be nothing while the network is functioning normally. However, when the administrator is annoyed, he will know that action is required.

Continuous processes may occur at constant intervals, analogous to crickets chirping at night. A common folk tale says 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) of a network host or service. A steady beep signifies that the host is alive.

Humans are very apt at recognizing when continual background sounds change, making problem detection with ease. If you read several books, chances are that you will not notice any change in your surroundings. However, if someone closes the door, you will immediately notice the change in sound. Peep takes advantage of this ability to notice sonic changes to represent a problem when all else fails. Through the absence of sound, Peep allows the administrator rest not fear about the network or computer. Peep operates, if it does, for the administrator's benefit.

The Peep system was designed to take advantage of existing system administration tools. Sniffer 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 distribution mechanisms such as CFengine [8, 7, 6] and rdist [9].

The Peep system has three major components: the *SoundServer* and *Peep* scripts, Each runs as a separate process. Communication between processes is addresses its original intended purpose and so must. This keeps client code simple, easy to debug, and easy to customize.

The *Peep* system was designed to be modular and utilize as little resources as possible. Our top priority client, *LogParser*, watches log files and uses regular expressions to determine when particular events have occurred. Because of the way regular expressions are compiled 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 be run on different machines to share the workload. Instances of *LogParser* communicate by sending messages to 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 network. Since log files are the standard communication channel between processes on a single machine, we felt that using them would be the most portable and flexible. Because we do not want to overload the system with excessive disk activity and disk-based bandwidth, *LogParser* can be configured to write its output to a ramdisk instead of a disk. Coupled with the single-machine method with its effect on the administrator's peace.

Expanding the capabilities of Peep to its 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-based hierarchy to define groups of clients that share the same sonic environment.

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 and flow control. However, Peep does not require reliable or ordered delivery of its messages. Reliability is unnecessary because the sound output should reflect the current state of the network and not require packets to be ordered in any way - so the packet transmission is best-effort - since the operation of the Peep system is dependent on 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 without affecting one another. We wanted users to be able to write their own clients with minimal knowledge of the underlying communication protocol. Because Peep's protocol is based on UDP, it is easier for users to write their own clients. We 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 heading mechanism with auto-discovery. This combination provides safe, dynamic, real-time bindings between clients and servers.

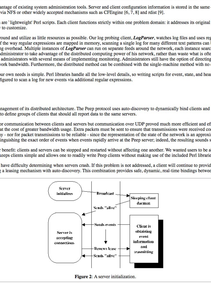


Figure 1: The Peep architecture.

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 sneaper without access to the Peep configuration file.

The Peep system was designed to take advantage of existing system administration tools. Sniffer 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 distribution mechanisms such as CFengine [8, 7, 6] and rdist [9].

The Peep system was designed to be modular and utilize as little resources as possible. Our top priority client, *LogParser*, watches log files and uses regular expressions to determine when particular events have occurred. Because of the way regular expressions are compiled 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 be run on different machines to share the workload. Instances of *LogParser* communicate by sending messages to 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 network. Since log files are the standard communication channel between processes on a single machine, we felt that using them would be the most portable and flexible. Because we do not want to overload the system with excessive disk activity and disk-based bandwidth, *LogParser* can be configured to write its output to a ramdisk instead of a disk. Coupled with the single-machine method with its effect on the administrator's peace.

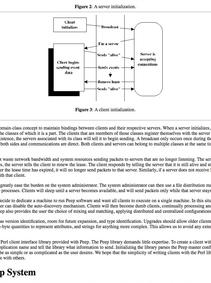


Figure 2: A server initialization.

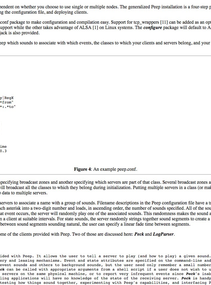


Figure 3: A client initialization.

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, when a client initializes, it broadcasts its existence to the subnets associated with its class. The servers that are members of those classes register themselves with the client and begin sending it packets. Conversely, when a client or server is terminated on both sides and communication ceases, the still-alive side notices the absence of packets from its counterpart and attempts to reestablish contact. Thus, the system can adapt to many servers concurrently.

Peep 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 it any more packets in the same manner. This allows us to avoid any crashes from representation issues, making the protocol more portable.

Details of the 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 anything else. 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 binary distribution, editing the master configuration file, and deploying clients.

The Peep server package uses the gnu configure package to make configuration and compilation easy. Support for tcp_wrappers [11] can be added as an option. Peep comes with two generic sound packages, one for Linux machines and one for Solaris machines. Compilation takes advantage of AULA [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 configuration, 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 mycluster
    broadcast 192.168.23.255:2000
    lease 60
    server sound.100hz
    client
client LogParser
    port 2000
    port 2001
    sensitivity
    regexp "Illegalinetaccess|Priority|Bogd
    sound [0] = "sendmail/1.+/hello"
    sound [1] = "sendmail/.+/.+/bye"
    sound [2] = "error"
    ramdisk /var/log/peep
    exec
    ethmon Type/Self,# sounds to load
    sound /path/sounds/peep/ping.+ 1
    ice-mail /path/sounds/peep/ice. 1
    sound /path/sounds/error/.+ 2 0.3
    end state
    ethmon Type/Self,# sounds: Fade time
    sound /path/sounds/error/.+ 2 0.3
    end state
```

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 may 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 provide for redundancy in the Peep network.

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 will then load any sound files that match the given pattern, up to the specified number. Sound files should be placed in directories by their name, so Peep will look in the directory for sound files that exist in subdirectories of the specified sounds. The optional fade-in parameter causes the sound to be faded in over the specified number of seconds. Sounds can be 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 these are discussed here: *Peek* and *LogParser*.

Peek

Peek 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. *Peek* is an example of a dumb client and bypasses the auto-discovery and leasing mechanisms. Event and state attributes are specified on the commandline and delivered directly to the server. Consequently, the server will play the sound specified by the commandline options. Peek's simplicity makes it ideal for testing new sounds. Peek's ability to be used interactively also lends itself to quick prototyping of client behavior. Peek's simplicity also allows it to be used in non-interactive scripts. For example, one might wish to write a script that should only ping *Peek* to talk to servers on the same physical machine, or to report very infrequent events *Peek*'s inability to use auto-discovery and leasing mean that it must be given the address of the server it is to talk to, and that it must be run periodically if it is to report a state. Thus, *Peek* is not suitable for all types of client-server communication, but it is a useful tool for debugging, testing sounds together, experimenting with Peep's capabilities, and interfacing Peep with other monitoring systems (such as an existing Nagios 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 also supports regular expressions compiled into memory. *LogParser* reads its entire configuration file at startup. It can be set to read and monitor various system logs, and can be instructed to send different messages to the server based on which pattern matches a given line. Because *LogParser* is designed to be multiple instances run on several different machines, each scanning for different sets of textual patterns in each client machine.

LogParser is flexible, easy to configure, and provides a clear and easy way to connect regular expressions to events and states. It analyzes log messages as they are written to the log file, and can be configured to watch for more than one single configuration syntax to generate command-line options and determine which sounds to associate with which particular events. Several options follow:

- The *port* option tell the server to listen on another port. This option must be given before any other options, or else the default port will be used instead.
- The *state* keyword tells the client, aside from pleasing the true collegiate, helps the user distinguish and even listen to sounds. Sonic location can even be used to convey information about the network. Imagine a Peep client that uses stereo or 4-way speakers to imply a client sound coming from the left, the right, the front or the back.
- 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 syntax. Others may find it difficult. The default is to match the entire line. The client will then send the entire line to the Peep server without being any more specific.

Directives in the *LogParser* configuration can be enabled or disabled via commandline options. Each line of the *LogParser* configuration corresponds to a unique, independent subprocess. In Figure 5, identical, but separate, mail-sounding rules are created from options "-f" and "-F", respectively. Thus, an invocation of *LogParser* searching for incoming mail might look as follows:

```
/usr/local/bin/LogParser -f /var/log/mail
    -sendmail "sendmail:.+:.+:.+"
```

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 leasing mechanisms. Multiple *LogParser* instances provide the user with a myriad of options for structuring the architecture of Peep within a network.

Peep Performance under Pressure

We dwelt with copious amounts of incoming network data, Peep has a queuing and windowing system that handles large numbers of simultaneous events. This section discusses the performance of Peep under such circumstances. Our goal was to ensure that Peep would not crash or become unresponsive when faced with a large number of events, and to allow the user to tune Peep's behavior under such circumstances.

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 configuration to limit the number of simultaneous sounds played at any given time. Sounds are played in the order they are received. Peep's queuing and windowing system also uses sound priorities to determine which sounds are important and should be played first.

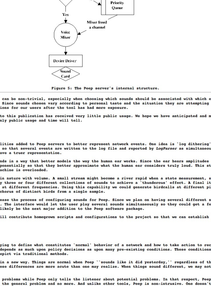
It is difficult to send events to a Peep server fast enough to fill a queue on a Pentium II 400 and during testing, this required the use of an infinite loop. If the queue is not filled, Peep spends most of its processing time waiting. It only fills the queue when a packet is received from the network. Thus, under light loads, the queue will place the events into a priority queue, ensuring that the user will play the most important events as soon as mixing channels free up. When the queue performs the processing necessary to fill the queue, it also performs other housekeeping tasks, such as checking for expired events and updating state sounds. The queue must also check the lease-time of each queued and current client sound, 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. Unfortunately, this means that Peep must do all mixing in software, which can be expensive. We felt that the ease of writing and distributing interpreted Peep Perl libraries outweighed the hardware. Future plans include support for hardware-based mixing as 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 and the difficulty of recording natural sounds, we have not been able to provide a large library of sounds that sound pleasant together. We have spent a great deal of time on the search of the perfect bird. Sounds we collected regularly contained other birds in the background and low-pass filtering and noise reduction, before they made five of enough internal background noise to serve as reset sounds. Collecting state sounds proved even more difficult, with the sound of wind being the most difficult. The challenge now is to collect 'desirable sounds' without violating fair use or laws and relative engineer.

In spite of the sensitive problems as the recording of natural sounds that we obtained from the Cornell Ornithology website [12], the Sidux Field Guide to Bird Sounds [13], and the Sound Dogs Sound Effects Library [14], we believe that Peep's ability to mix sounds together to create a pleasant listening of sounds included with Peep. Nevertheless, we have made significant progress in providing a Wellsonds venue, and are planning others in the future.


The Listener
Resulted Packet
Sound Mixer
Network Listener Thread
Sniffer Thread
Sniffer Thread
Engine (only Sniffer Thread)
Is Sniffer Alive?
Yes
No
Relaunch Sniffer
Sniffer Thread
Sniffer Thread
Sniffer Thread