

**streaming consciousness (and audio)**  
**or,**  
**If On A Winter's Night A Squeaker**  
**(with apologies to a certain Italian fiction writer)**

*Craig Latta*  
*The NetJam Project*  
*netjam.org*

---

netjam.org/flow

It's been a slow day. Browsing through the day's mail over a fresh cup of coffee, you notice your copy of "Squeak: Applications and Community/Taking Another Path" has arrived. You peruse the covers lazily, noticing that a chapter about streaming audio lies within.

Streaming audio!

Feverishly, you scan the table of contents. A shudder runs down your spine as your eyes alight on the title and the page number is revealed. The pages fly in a flurry of acid-free abandon, finally revealing page xxx. There you read, in letters which shall remain indelibly etched in your psyche, "netjam.org/flow".

This is all you need to see; there's not a moment to lose. Instantly the book is forgotten as your fingers pound the URL into the nearest keyboard, and instructions flood the screen. A few clicks later, installation is complete. Breathlessly, you... DO IT:

```
(Sound fromHostNamed: 'netjam.org') play
```

*(figure one, sound controls)*

A small panel of sound-playing controls appears; a play button activates. One internet moment later, the strains of a new yet somehow familiar song sift forth from your speakers. You experience a state of giddy bliss. Streaming audio! In Squeak! A verse goes by, then a chorus. Then another verse. Another chorus. Then... another verse followed by another chorus. But then a bridge! Can a break be far behind?

*(figure two, a notifier with the title "break")*

Indeed, the break has come quite literally. The music has stopped. A notifier stares innocently at you. It's all too much; utterly spent, you pass out.

---

Some time later, you regain consciousness. An eerie silence fills the room in the twilight, broken occasionally by the gentle swapping of hard disks. Then it hits you. The song... the song was trying to tell you something.

---

## streaming consciousness (and audio)

You've never really explored the implementation of audio or networking before, but it doesn't matter. You simply must hear the rest of that song. Summoning a new reserve of energy, you reach for the mouse. With a click the notifier gives way to a debugger. The contexts of the suspended process lay before you.

*(figure three, detail showing the debugger's context list)*

The most recent one deals in receivers you've never seen and messages you've never sent. You decide it's best to start at the beginning, if you're going to understand this. There's `BlockContext>>newProcess`, of course, where the process was created. Things get down to business in `(SoundPlayer class)>>startPlayerProcessBufferSize:rate:stereo:.` You notice that this method stores the process in a variable, under the name `PlayerProcess`. And you notice a distinct lack of the usual human interface machinery in the selectors of the other contexts. This must not be the human interface process. This makes sense; sounds are played from a dedicated process, so that the other processes may continue unaffected.

You also notice that this method determines the rate at which audio data is played. You realize an important terminological detail: the difference between "sample" and "frame" as they are used here. A sample is an instantaneous measure of a sound waveform's amplitude from a single position. A frame is a collection of concurrent samples for one or more waveforms, each of which is transmitted over a "channel". Your sound is in stereo, so it has two waveforms (for the positions of the left and right ears), and thus two samples for each two-channel frame. The data is played at a certain "frame rate", also known (less accurately) as the "sampling rate".

*(figure four, a diagram illustrating the terms defined above)*

You venture further, into `(SoundPlayer class)>>playLoop`. Here you see the connection between the sound-playing process and your own beloved sound. `SoundPlayer` asks each active sound to mix an amount of pending audio data into a buffer, then plays the buffer. It's your sound's turn to mix, in response to `mixFramesInQuantity:into:startingAt:leftVolume:rightVolume:.`

*(figure five, a diagram illustrating the sound-playing process)*

You notice that your sound is an instance of `NetSound`, a subclass of `StreamingSampledSound`. Since the mixing message is sent to each active sound, it must be implemented by each sound class. A quick look at the implementors of the mixing message highlights a significant difference between `StreamingSampledSounds` and sounds of other classes. Whereas the other kinds supply their audio samples algorithmically (as with `FMSound`), or from static caches (as with `SampledSound`), `StreamingSampledSounds` supply them from a dynamic cache, fed by a "sample-stream". You move forward a context to `NetSound(StreamingSampledSound)>>fetch:.`, where your sound's cache is replenished.

The `NetSound`'s sample-stream is a `NetStream`. You know a bit about streams-- the system transcript is one you use often. The basic idea is pretty simple: a stream provides interleaved reading and writing access to a collection, with messages derived from `next` and `nextPut:.` But the

---

## streaming consciousness (and audio)

NetStream's collection probably isn't a typical internal collection, since, as its class name implies, its elements come from another machine on the Net.

You look next into `NetStream>>next:into:`, which leads to `NetStream>>next:into:startingAt:timeoutAfter:`. There, you see that the NetStream uses a "resource" instead of an internal collection. At long last, the next context is the one at which the player process is suspended. Here you see that the resource is an instance of `OutgoingClientTCPSocket`, a subclass of `NetResource`. This must be where the actual bytes come from. But looking at the instance creation protocol of NetStream, you realize that the resource needn't be an "outgoing client TCP socket", or even a socket at all. NetStream can create instances with several different resources. Browsing the `OutgoingClientTCPSocket` class, you notice it's a subclass of `NetResource`, which has other subclasses, including `UDPSocket` and `SerialPort`. With the same NetSound and NetStream interface, your audio frames could just as well come from a serial port or an infrared link.

*(figure six, a diagram illustrating the relationship between NetStreams and external resources)*

---

Night has fallen. The room around you has vanished, eclipsed by the glow of the screen. You turn on a torchiere and take stock of your surroundings... There's a dedicated sound player process which continually fills a buffer with audio data, then plays it. It fills the buffer by asking each active sound to mix some quantity of its ensuing frames with the sound buffer's current contents. Your sound gets its frames from a sample-stream, which has as its source a socket connected to a remote machine. A dog barks from somewhere in the distance. Your coffee has long since gone cold. There are four messages waiting on your answering machine.

*(figure seven, an elaboration of figure five, showing the network data components mentioned above)*

This is all very well, you think, but how does the NetSound know how many bytes to get from its NetStream? How did the sound know what its frame rate was supposed to be? How does it know the number of channels to use, and how many bytes are used by each frame? In the case of other sounds, this information is either implicit (as with FM sounds), or specified "out of band" in a header (as with sounds played from files). Perhaps NetSounds have a means of communicating out-of-band information about sound in play...

You go back a few contexts, where your NetSound is the receiver, and look at the instance variables. Among them is the promisingly-named "control". Clicking on it for a description, you read that it's "an `AudioClient`, connected to port 7777 of netjam.org, controlling a paused sound". This is what you're after. Instinctively, you start to open a browser on `AudioClient` when a word jumps out at you from the screen: "paused". This `AudioClient` knows something about your sound (or thinks it does). Your hand unfreezes and the browser is open.

The first thing you notice about `AudioClient` is that it seems to be part of a larger client/server framework. It's a subclass of `Client`, which has `Server` for a sibling and `Correspondent` for a parent. `Client` and

---

## streaming consciousness (and audio)

Correspondent seem to handle the details of connecting to a server and setting resource options (such as whether to communicate in binary or text). Having created an `AudioClient` with `>>toHostNamed:`, a `NetSound` is free to start using it as a sort of remote control for audio. This is just what happens in (`NetSound` class)`>>fromHostNamed:`, the first method you ran. `NetSound` creates an `AudioClient` as a control, selects a track to play, then creates an instance of itself with a sample-stream, channel quantity, frame size, and frame rate obtained from the control.

Looking at the instance behavior of `AudioClient`, you see how an instance answers these parameters. Using its `NetStream`, it sends command numbers and answers objects based on the results. For the numerical answers, the server result itself is used. In the case of the sample-stream, the client answers a new `NetStream` connected to the remote port indicated by the server result. These commands form the basis for a very simple audio control protocol.

If there's an `AudioClient`, might there be an `AudioServer` as well? Sure enough, there is; a quick browse through it confirms the behavior suggested by `AudioClient`. For each client connected to it, an `AudioServer` keeps a reference to a sequence of audio frames (typically on local secondary storage), and a `NetStream` onto which it writes them. Concurrently with writing frames, an `AudioServer` listens for commands from each client. Some of these commands request changes in the writing of the frames.

*(figure eight, an elaboration of figure seven, showing the network control components mentioned above)*

In particular, you notice `AudioServer>>pause`. Here, the server simply stops writing frames, keeping its current position in the local frame sequence. Obviously, this would stop the flow of frames to the client machine. Perhaps this is what happened with your sound... you feel yourself getting closer to an answer here. With a raised eyebrow you turn your attention to `AudioClient>>printString`, the method which printed the tell-tale description. Here, the client uses an instance variable named "paused". Surveying other references to that variable, you see `AudioClient>>pause`. As you expect, it sends a pause command to the server. However, in the blank space below the end of method you see...

A bug!

Sending the pause command to the server is all that `AudioClient>>pause` does. This merely shuts off the flow of audio frames. It should also take your sound out of the sound player's queue of active sounds. After all this browsing, you finally get to write some code of your own. `AudioClient` specifies an instance variable, called "sound", which refers to the sound whose frames it supplies. In a brief burst of keystrokes, you add "sound pause" to `AudioClient>>pause`, and recompile the method. You're eager to try out your fix; you're about to close the debugger and try playing the sound again.

But then you realize: an intentional pause in the flow of audio frames isn't the only reason the frames might stop. A simple lapse in connectivity could cause this as well. And regardless of the cause, the system really ought to do something more graceful than open a debugger when this happens. You return to the most recent context of the suspended process in

---

## streaming consciousness (and audio)

the debugger. It reveals that execution stopped due to an unhandled exception:

*(figure nine, detail of the debugger's source pane, showing the expression "thisContext handle: #networkTimeout", perhaps alpha-highlight)*

What's needed here is an exception handler. Browsing through the senders of `>>handle:`, you come across several examples in `BlockClosure`. The basic idiom is to do the risky work inside a block, sending `#valueHandling:with:` to it. You change `StreamingSampledSound>>fetch:` so that reading from the `NetStream` retries if a network timeout occurs:

*(figure ten, the replacement code mentioned above, done as a figure because Word won't let me format it the way I want it)*

It's still not the most flexible solution, but it'll do for now.

Now that you no longer need the debugger, you close it. It's time for the moment of truth. Fine beads of sweat trickle across your forehead as you evaluate the initial expression again... The music begins once more, pulsing hypnotically. A great sense of foreboding overcomes you as the dreaded break approaches. And then-- the music pauses, without a notifier in sight. Instead, the pause button on the sound's control panel illuminates. You merely press it, and the song continues. Success!

---

You were right. The song *\*was\** trying to tell you something. You listen attentively, nodding thoughtfully. Truer words you've never heard. That was certainly worth waiting for. Having reached a state of closure, you head sleepily toward bed. On your way, you come across the Squeak book, still open to the streaming audio chapter. Perhaps tomorrow you'll read it.