Sharing Time and Code in a Browser-Based Live Coding Environment

Charlie Roberts
University of California at Santa Barbara
charlie@charlie-roberts.com

Karl Yerkes
University of California at Santa Barbara
yerkes@mat.ucsb.edu

Danny Bazo
University of California at Santa Barbara
dannybazo@mat.ucsb.edu

Matthew Wright
University of California at Santa Barbara
matt@create.ucsb.edu

JoAnn Kuchera-Morin
University of California at Santa Barbara
jkm@create.ucsb.edu

ABSTRACT

We describe research extending the live coding environment Gibber with affordances for ensemble, networked, live coding performances. These include shared editing of code documents, remote code execution, an extensible chat system, shared state, and clock synchronization via proportional-integral control. We discuss these features using the framework provided by Lee and Essl in their 2014 paper *Models and Opportunities for Networked Live Coding*.

1. Introduction

Collaborative editing of documents has been a staple of browser-based authoring tools for many years, with operational transforms (Ellis and Gibbs 1989) enabling writers to freely edit documents with other authors concurrently. Concurrent editing of shared documents is especially promising for ensemble live-coding performance, as it offers performers the ability to view and edit the work of their fellow performers, but it also creates a variety of challenges. Having implemented concurrent editing of code in two different environments for live coding performance, we share lessons learned and outline feedback mechanisms to help inform ensemble members when code is executed during performance and who executed it. In addition, we describe research that automatically synchronizes Gibber instances to a master rhythmic clock via PI control structures, even with the inherent jitter imposed on communications by the TCP-based WebSocket protocol.

Our research enables a group of performers to sit in a room with their laptops, join a network, execute a single line of code, and immediately enter an ensemble performance where each performer's code is freely available for editing by every member and all clocks are synchronized. We call this system *Gabber*; it works inside of the browser-based live coding environment *Gibber* (Roberts and Kuchera-Morin 2012).

2. Strategies for Networked Ensemble Performance

Lee and Essl survey strategies enabling networked live coding performances in (Lee and Essl 2014a). Gibber incorporates many of these strategies; this section builds off their work by describing Gibber's networked ensemble affordances using their terminology.

2.1. Time Sharing

The first strategy discussed by Lee and Essl is *time sharing*, which concerns synchronizing events between performers. Although Gibber initially had no methods for sharing time, we have addressed this problem by phase-syncing all schedulers in ensemble performances as described in Section 3. Despite the research outlined here on temporal synchronization, we note that there is a rich terrain of performance possibilities that can be explored without synchronization; three authors on this paper have successfully given a number of such performances (Roberts et al. 2014).

2.2. Code Sharing

Gibber implements code sharing at a number of different levels. For asynchronous collaboration, Gibber provides a centralized database enabling users to save and retrieve the programs (aka giblets) they create. This database is searchable, and users can provide a variety of metadata about giblets to aid in the search process. Giblets can be easily iterated and a versioning system enables users to examine changes throughout a giblet's development. Most importantly, every giblet can be accessed via a unique dedicated link, providing an easy mechanism for users to distribute their work to others.

Gibber's initial synchronous affordances enable any registered Gibber user to start collaboratively editing a code document with a remote user (who can be located anywhere with internet access) by simply clicking their name in one of Gibber's chatrooms; this feature also enables users to remotely execute code across the network if permission is granted. In this instance, users can either find each other through happenstance by joining the same chat room, or users who know each other can agree to meet in a particular chat room to participate in a shared editing session together. We envision the second instance being particularly useful for classes taught in computer labs or via distance learning; a teacher can have all students meet her/him in a particular chat room and then assist students as needed via the collaborative editing features.

Although performances consisting of a single, shared code editor initially seemed promising, in practice they proved problematic (Wakefield et al. 2014), as program fragments frequently shifted in screen location during performances and disturbed the coding of performers. A better strategy is to provide a dedicated shared editor for every member of the performance; this strategy was adopted in the *Lich.js* live coding environment (McKinney 2014) with read-only code editors. In other words, with N players there are N collaboratively editable documents, along with a social convention that each document uniquely "belongs to" or is somehow primarily associated with a single person. Our implementation of Gabber provides this capability, enabling everyone to code in their own personal editor and also read and edit the code of other users as necessary.

Although Gibber's original code-sharing model allowed this, quadratic growth of managing code sharing permissions made it too cumbersome for $N > 2$. We subsequently implemented a much simpler method for beginning networked ensemble performances in Gibber. After performers agree upon a unique name for a Gabber session, they begin the session with a single line of code:

```
Gabber.init( 'uniqueSessionName' )
```

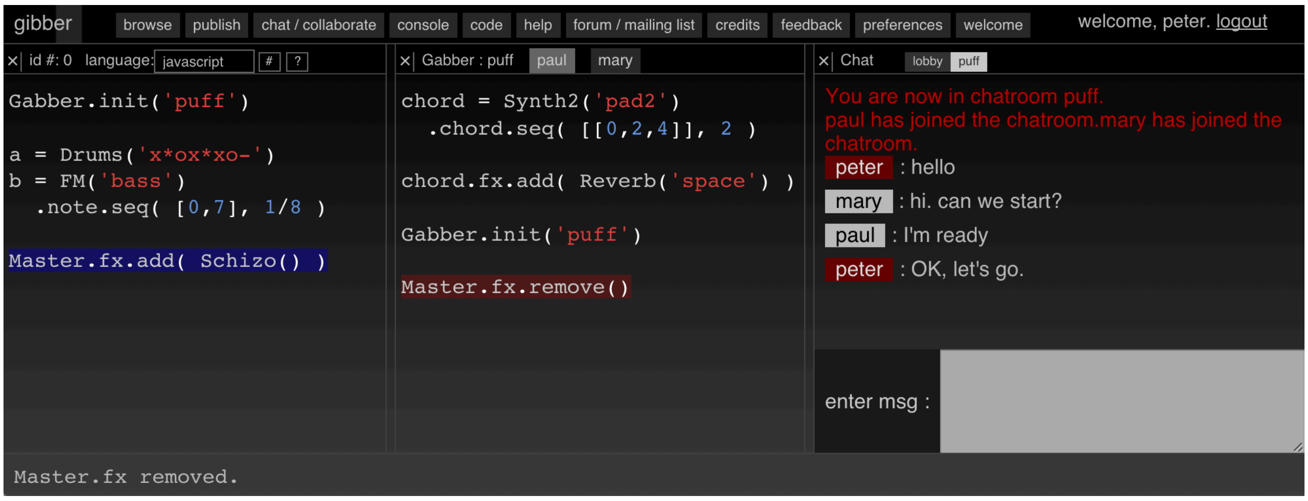
Executing this either creates (if necessary) or joins (if already existing) a named networked live coding session on the worldwide Gibber server, and triggers the local browser to:

1. Open a Gibber chatroom named identically to the Gabber session that contains all the session participants.
2. Create a new user interface column containing everybody else's shared code editors (and updating dynamically as people join or leave).
3. Share the code editor where the call to `Gabber.init` was executed with all members of the session.

After these actions are carried out, the user interface of the Gibber environment will resemble Figure 1. Clicking on a tab in the shared editing column reveals the associated user's code editor.

2.3. State Sharing

There are two modes of state sharing in a Gabber session. The default *local* mode is designed for physically co-located performances: each user explicitly selects code (from anybody, as described in Section 2, although performers will commonly execute code they authored) and triggers execution on their own computer to generate a unique audio stream which renders only their personal actions. Users can also explicitly execute code on the machines of other ensemble members using specific keystroke macros. If the executed code is contained in personal editor of the executing performer then it is broadcast and run on all machines on the network. If instead the code is contained in the shared editor



```
Gabber.init('puff')

a = Drums('x*ox*xo-')
b = FM('bass')
  .note.seq( [0,7], 1/8 )

Master.fx.add( Schizo() )
```

```
chord = Synth2('pad2')
  .chord.seq( [[0,2,4]], 2 )

chord.fx.add( Reverb('space') )

Gabber.init('puff')

Master.fx.remove()
```

You are now in chatroom puff.
paul has joined the chatroom.mary has joined the chatroom.
peter : hello
mary : hi. can we start?
paul : I'm ready
peter : OK, let's go.

```
Master.fx removed.
```

Figure 1: *A screenshot of the Gabber interface in a networked performance with three performers: Peter, Paul and Mary. The screenshot shows what Paul is seeing. Note the two highlighted code fragments; the blue highlight indicates code that the performer (Peter) has sent over the network to be executed. The red highlight indicates code that another member (Paul) of the ensemble has remotely executed on Peter's machine.*

of another performer it is only executed on the associated performer's computer, enabling, for example, a user to lower the amplitude of a particular ugen they find to be too loud.

In contrast, *remote* mode assumes performers are in physically disparate locations; this in turn means that each computer must locally render the entire performance. Executing any code in remote mode automatically also sends it across the network for execution by all ensemble members. Although prior research has streamed audio between ensemble members (Wang et al. 2005), remotely executing code offers a much lower bandwidth footprint, but comes with potentially higher CPU usage for each computer and the inability to share stochastic or interactive components of a performance.

In order to mitigate this, Gabber provides an easy mechanism for sharing arbitrary JavaScript data, drawing inspiration from the tuple approach adopted by *Impromptu* (Sorensen 2010) and the shared namespace used in *urMus* (Lee and Essl 2014b). The main `Gabber` object contains a `shared` data object which is distributed among all members of a Gabber performance. Defining a shared variable consists of a single method call; after executing this method any changes to the variable are distributed to all ensemble members. An example of defining a shared variable and changing it is given below:

```
Gabber.shared.add( ‘pattern’, [ ‘c4’,’d4’,’g4’ ] ) // define & distribute the initial ‘pattern’ variable
Gabber.shared.pattern = [ ‘d4’,’d#4’,’g#4’ ]       // change the pattern and distribute changes
```

Importantly, properties of the PI controller used to synchronize the clocks of Gibber instances (discussed in Section 4) are all properties of this shared object, enabling users to easily tune the controller and automatically propagate changes to all participants.

2.4. Access Control and Catching Them Red Handed

Although various strategies for limiting code execution on the machines of other ensemble members have been explored (Lee and Essl 2014b), Gibber uses a simpler, fully permissive scheme similar to the *Republic* quark for SuperCollider (Rohrhuber et al. 2007). Remote code execution, while presenting powerful opportunities for spatialized code execution, also has great potential for misuse and mischief. In performances by the CREATE Ensemble at UC Santa Barbara, one notorious user named `killbot` would routinely start algorithms running on remote machines that would gradually kill off running unit generators. As a result, designing killbot-proof algorithms became a game for the ensemble. A call to `Gabber.block( 'userNameToBlock' )` blocks remote execution by a particular performer, removing the challenge of blocking killbot but perhaps making for less stressful ensemble dynamics.

Even assuming that ensemble members have each others' best interests at heart, there is a need for performers to understand when (and by whom) code is remotely executed on their computers, e.g., so performers understand sudden changes in the sounds they are generating. For this reason we have added a number of simple visual annotations to Gibber to indicate remote code execution in *local* mode, as shown in Figure 2. When code is executed and sent to all members of a performance, it is highlighted in red in all editors. When code is remotely executed on the computer of a single performer it is highlighted in blue.

2.5. Communication Facilitation

Each Gabber session has an associated chatroom for coordinating performance structure, discussing code, and banter. There are also programmatic hooks enabling users to detect when chat messages arrive and take action based on their content and/or author. These hooks have been used in summer camps for high school students to explore generative music-making in a program run by the Experimental Music and Digital Media program at Louisiana State University, and in classes at UC Santa Barbara.

3. Centralized and Decentralized Network Structures

Lee and Essl define a *centralized* performance network as one where clients submit code to a server for rendering. The first ensemble Gibber performance, performed in April of 2012, adopted this model (Roberts and Kuchera-Morin 2012; Roberts et al. 2014): a single server computer drove both the audio and visual projection systems, with scrolling displays of all code sent from clients and of the performers' chatroom. As Lee and Essl describe, this model automatically synchronizes time and state according to when the server executes each received code block. One additional benefit is the potential to preview code changes before submitting them to the server. Gibber's different keystroke macros to execute code locally vs. remotely enable performers to audition code for execution on headphones before sending it to the server for projection through the house speakers. This style of previewing only works in performances where each ensemble member is submitting code that does not rely on the state of the master computer; accordingly, in our first Gibber performance each member had particular ugens that they were responsible for creating and manipulating. Gabber can easily mimic these early performances. Ensemble members can use headphones on their personal laptops while a single, central, computer is connected to the audiovisual projection systems. Users target their personal computer (to audition) or the central computer (to present to the audience) for code execution as described in Section 2.3

Research presented in this paper extends Gibber to support the other two network models discussed by Lee and Essl. The *remote* mode of performance, described in Section 2.3, supports the *decentralized* model, where every machine runs all elements of the performance concurrently. The *local* mode of performance supports *decentralized programming with central timing*, enabling users to have personal control over state on their computers while providing temporal synchronization.

4. Network time synchronization

Many network time synchronization solutions exist (e.g., GPS, NTP, PTP), but most are implemented at the system level, requiring software installation and careful configuration of each machine in the network. Furthermore, these common solutions only maintain synchronization of system clocks (which Gibber does not use), not audio clocks (which Gibber does use). By implementing network time synchronization entirely within Gibber, we free users from the need to run external software to achieve synchronization. In effect, setting up a shared performance requires nothing more than an internet connection enabling users to connect to the worldwide Gibber server. The Gibber server is freely available on GitHub if users need to run their own local instance; this could be useful in a performance where an internet connection is unavailable or unreliable.

4.1. Synchronization with a Hybrid Proportional Controller

The Gabber server broadcasts its time (in samples) to all Gibber instances (clients) on the network via TCP/WebSockets once per audio buffer (typically 1024 samples). Upon receiving the server's time (now out-of-date because of network latency), each client subtracts its local time. Controller code within each client uses this difference (aka error) to decide how to correct the client's local time such that the client's timer converges to synchronize with that of the master.

We use a hybrid controller (Lygeros and Sastry 1999) with two modes: coarse and fine. The coarse mode operates when the client is out of sync beyond a particular threshold, and immediately sets the client's time to the most recently received master's time, disregarding any discontinuities this causes. For example, the first time a Gibber instance receives the server's time (when someone joins a performance already in progress), the coarse mode takes effect immediately, getting the new performer roughly in sync with the rest of the group. Audio dropouts caused by spikes in CPU usage in the client can also cause this type of correction.

The fine mode, which operates when the error is below the threshold, uses a proportional controller to make small, smooth adjustments to the client's time. The default proportional constant was chosen empirically following experimentation and observation but can be freely edited by users; as described in Section 2.3 such changes to the controller are immediately propagated to all members of a performance. The controller acts on a low-passed version of the error signal due to the relatively low rate at which the error calculation and control is performed (typically 44100/1024 Hz, once per audio block callback). Changing the length of the running mean involves a tradeoff between fast response (short mean length) and smooth operation but long convergence time (long mean length).

Our nominal value of the threshold for switching control modes was also determined empirically through consideration of the minimum amount of time that a client would have to be out of sync for the coarse correction to be more aesthetically pleasing (approximately 100 ms for 120bpm 44100Hz sample rate group performance, varying according to musical features such as tempo). The threshold used for coarse correction is exposed to the user via Gabber's shared data object.

The code below is our controller implementation in JavaScript pseudocode:

```
K_p = .05
coarseThreshold = .1 * sampleRate
runningMean = RunningMean( 50 )
localPhase = 0

function onMessageFromServer( masterPhase ) {
  error = masterPhase - localPhase
  if( abs( error ) > coarseThreshold ) {
    // coarse controller
    localPhase = masterPhase
    Gibber.setPhase( localPhase )
    runningMean.reset()
  }else{
    // fine controller
    var phaseCorrection = K_p * runningMean( error )
    localPhase += phaseCorrection
    Gibber.adjustPhaseBy( phaseCorrection )
  }
}
```

(Brandt and Dannenberg 1999) presents a method of "Forward-synchronous" audio clock synchronization using PI-control. Their implementation accounts for noisy error measurement using a second PI controller that takes advantage of local, low-jitter system clocks on clients. We use signal conditioning (moving average) instead. Their system makes control adjustments much less often than ours.

(Ogborn 2012) describes EspGrid, a protocol/system for sharing time and code, as well as audio and video among players in networked laptop orchestras. EspGrid uses Cristian's algorithm (Cristian 1989) to present a synchronized system clock to each user so that s/he might schedule the playing of shared beat structures. We implemented Cristian's algorithm in Gibber, but found the results unsatisfying compared to our current solution. EspGrid has the benefit of using UDP while Gibber is limited to TCP/WebSockets (and the jitter associated with its error correction) for network messages. This could be one reason for our comparatively subpar results with Cristian's algorithm.

4.2. Results

In tests, clock synchronization in Gibber has produced aesthetically satisfying results. Sound events feel like they happen "in beat" even on a WiFi network, with users experiencing quickly fixed audio glitches. However in unfavorable WiFi conditions (i.e., heavy traffic, many overlapping networks, interference) the controllers in Gibber instances may exhibit thrashing and/or non-convergent correction. By adjusting the proportional control constant, coarse mode threshold time, and error signal low-pass length, the system can be made more tolerant of various differing conditions, but as a general rule it is best to use a wired LAN in performances where wireless conditions are less than ideal.

When a Gibber instance becomes "out of beat" for whatever reason (i.e. network problems, jitter, high CPU load, etc), it typically heals within the coarse mode threshold time. In our tests, measurements of the steady-state error for clients on a WiFi network had a mean of 13.5 samples with a standard deviation of 39.2 samples. For comparison, the measured network jitter was approximately 2ms (~88 samples).

5. Conclusions and Future Work

We have described an extension to Gibber, named Gabber, that significantly eases setup of networked live coding performances. Joining a Gabber performance requires only a single line of code to be executed, and provides clock synchronization, code and state sharing, remote execution of code, and an extensible chat system for networked performances.

In the future, we are particularly interested in additional visual indicators documenting the actions of individual performers to their fellow ensemble members. For example, given that a performer is responsible for creating and manipulating a particular set of ugens, a waveform and/or spectral visualization of their summed output could be displayed in the associated tab of their shared editor on remote machines. This would enable users to, at a glance, see the overall spectrum and amplitude contributions of each member, potentially giving them an idea of who is responsible for each instrument.

Further future work will be informed by performances given using Gabber. Although the CREATE Ensemble has successfully performed a number of networked pieces using Gibber (as has the Laptop Orchestra of Louisiana) we have yet to perform with Gabber in its current incarnation. We are excited about the introduction of a shared clock for ensemble performances, and the resulting possibilities for networked algorave participation.

6. Acknowledgments

We'd gratefully acknowledge the Robert W. Deutsch foundation, who made this research possible.

References

Brandt, Eli, and Roger B Dannenberg. 1999. "Time in Distributed Real-Time Systems."

Cristian, Flaviu. 1989. "Probabilistic Clock Synchronization." *Distributed Computing* 3 (3): 146–158.

Ellis, Clarence A, and Simon J Gibbs. 1989. "Concurrency Control in Groupware Systems." In *ACM SIGMOD Record*, 18:399–407. 2. ACM.

Lee, Sang Won, and Georg Essl. 2014a. "Models and Opportunities for Networked Live Coding." In *Proceedings of the Live Coding and Collaboration Symposium.*

———. 2014b. "Communication, Control, and State Sharing in Networked Collaborative Live Coding." In *Proceedings of the 2014 Conference on New Interfaces for Musical Expression*, 263–268.

Lygeros, J., and S. Sastry. 1999. "Hybrid Systems: Modeling, Analysis and Control." UCB/ERL M99/34. Electronic Research Laboratory, University of California, Berkeley, CA.

McKinney, Chad. 2014. "Quick Live Coding Collaboration in the Web Browser." In *Proceedings of the 2014 Conference on New Interfaces for Musical Expression*, 379–382.

Ogborn, David. 2012. "EspGrid: a Protocol for Participatory Electronic Ensemble Performance." In *Audio Engineering Society Convention 133.* Audio Engineering Society.

Roberts, C., and J.A. Kuchera-Morin. 2012. "Gibber: Live Coding Audio in the Browser." In *Proceedings of the International Computer Music Conference.*

Roberts, Charles, Matthew Wright, JoAnn Kuchera-Morin, and Tobias Höllerer. 2014. "Gibber: Abstractions for Creative Multimedia Programming." In *Proceedings of the ACM International Conference on Multimedia*, 67–76. ACM.

Rohrhuber, Julian, Alberto de Campo, Renate Wieser, Jan-Kees van Kampen, Echo Ho, and Hannes Hölzl. 2007. "Purloined Letters and Distributed Persons." In *Music in the Global Village Conference (Budapest).*

Sorensen, Andrew C. 2010. "A Distributed Memory for Networked Livecoding Performance." In *Proceedings of the ICMC2010 International Computer Music Conference*, 530–533.

Wakefield, Graham, Charles Roberts, Matthew Wright, Timothy Wood, and Karl Yerkes. 2014. "Collaborative Live-Coding Virtual Worlds with an Immersive Instrument." In *Proceedings of the 2014 Conference on New Interfaces for Musical Expression.*

Wang, Ge, Ananya Misra, Philip Davidson, and Perry R Cook. 2005. "CoAudicle: a Collaborative Audio Programming Space." In *In Proceedings of the International Computer Music Conference*. Citeseer.