Communication, Control, and State Sharing in Networked Collaborative Live Coding

Sang Won Lee
Computer Science and Engineering
University of Michigan
2260 Hayward Ave
Ann Arbor, MI 48109-2121
snaglee@umich.edu

Georg Essl
Electrical Engineering & Computer Science and Music
University of Michigan
2260 Hayward Ave
Ann Arbor, MI 48109-2121
gessl@umich.edu

ABSTRACT

In the setting of collaborative live coding, there emerges a number of issues: (1) the need for communication, (2) issues of conflicts in sharing program state space, and (3) remote control of code execution. This paper proposes solutions to these problems. In the recent extension of *UrMus* - a programming environment for mobile music application development- we introduce a paradigm of shared and individual namespaces safeguarded against conflicts in parallel coding activities. We also develop a live variable view that communicates live changes in state among live coders. Lastly, we integrate collaborative aspects of programming execution into built-in live chat, which enables not only communication with others, but also distributed execution of code.

Keywords

live coding, network music, mobile music, collaborative music making

1. INTRODUCTION

In this paper, we discuss various extensions to a live coding environment in order to facilitate aspects of networked collaborations. Specifically, these extensions are (1) introduction of individual and shared namespaces in conjunction with a live variable view to help networked live coders manage and visualize program state, and (2) a live coding chat environment that supports remote code execution among live coders as well as non-coding instrument performers.

Music is inherently a communicative and collaborative art among composers, performers, instrument builders and listeners. The same can be said for live coding, a crucial aspect of which is communication among live coders, instrument performers and the audience. Further, the presence of networked programmers poses challenges of managing possible programming conflicts that might arise from misunderstanding or miscommunication. The goal of this work is to provide structural support within the live coding editing environment as well as in live-coded artifacts (e.g.mobile apps) support that mitigate these risks by providing critical information or explicit control.

2. COLLABORATIVE LIVE CODING

We can define collaborative live coding as live coding practice that involves more than one musician performing, in a collaborative manner, live coding music (see some conceptual examples in Figure-1). Such collaboration enriches the range of the live coding music's expressivity just as if musicians were playing as an ensemble. A mode of collaboration in live coding can be group improvisation where two musicians perform, respond to each other based on what they hear at the moment and shape music into a coherent piece. In this mode, live coders are connected through feedback loop created in sonic space, depicted in Figure 1-(a).

2.1 Live coding as Network Music

Collaborative live coding does not necessarily require network capability. However, the fact that live coders play on a laptop invites a variety of techniques that have been explored in the field of network music, given that network capability is a very basic function of modern computers. The potential of live coding as network music was anticipated from the inception of live coding [4].

Network music researchers have proposed ways to classify networked music-making environments; live coding could benefit from applying the dimensions used in classifying network music. Indeed, doing so could shed light on and expand mode of collaborative live coding. Barbosa proposed a way to classify computer-supported collaborative music based on synchronisms (*synchronous / asynchronous*) and tele-presence (*local / remote*) [1]. Weinberg classified interconnected music environments according to the roles of computers and the level of interconnectivity among users [15]. He also described a number of topologies depending on relations among musicians (*centralized / decentralized*) and the nature of interconnectivity (*synchronous / sequential*). We borrow these concepts directly to understand existing works of collaborative live coding and to associate with our design choices in the rest of the paper.

2.2 Networked Collaboration within Live Coders

Many previous works have attempted to augment live coding music performance with network capability implemented via live coding language and its programming environment. In a live coding scenario, data transmitted over a network need not to be audio. Rather, code text (or program state) is used as elements of a network conversation and sound can be displaced from the source by executing, remotely, transmitted code. *PowerBook Unplugged* exemplifies an interconnected and *decentralized* network ensemble by sharing code over a wireless network among live coders [12]. Similarly, *aa-cell* added network capability to *Impromptu*, to facilitate collaboration over the *local* and *decentralized* network for sharing and executing code remotely [2]. In a broader sense, *reacTable* can count as a collaborative live patching environment where multiple users can build a sound synthesis patch with shared tangible objects in a *local* and *centralized* manner [8].

Aside from sharing and exchanging music (or code) over the network, it is often times helpful during the session to coordinate temporal synchronization (e.g., tempo, meter, clock) between live coders. Sorensen introduced *Impromptu Spaces* and take a *decentralized* approach to implement a distributed memory that can be read and modified from multiple computers [13]. The *Spaces* also provides a blocking procedure, which is useful for clock synchronization. It follows the server-client structure in which one feeds synchronized temporal information (e.g., metronome) and the other processes use the primitives locally.

A course of works have been designed and developed for collaborative live coding. Wang introduced a concept of collaborative audio programming space called Co-Audicle, and suggested design frameworks for it [14], although we cannot find that the system has ever been implemented to fulfill a networked and collaborative live coding scenarios. While it offers no fully-functional programming language as do other live coding languages, Freeman's *LOLC* is designed for large-scale, collaborative improvisation based on a simple command-line-like language [6]. It takes a hybrid strategy of utilizing both the *decentralized* structure for sound generation (each machine plays its own sound) and the *centralized* structure for providing a shared sonic environment. The dual server-client applications accommodate large-scale laptop ensembles and the server application provides a shared environment for code exchange, instant chat, distribution of musical patterns, and a synchronized global clock for client applications. The server application chooses to selectively visualize the state of collaboration rather than project the computer screens of all players.

Based on Barbosa's classifications, many of the aforementioned examples fall into the *local/synchronous* category. However, *Gibber* enables a live coding environment on a web browser so that multiple live coders online can collaborate in *remote* setup by sending code and remotely executing it [11]. The most recent version of *Gibber*[1] includes a Google docs-like editor where one can write code and read code of others in real-time, enabling multiple performances at the same time if two collaborators are in different geo-locations. It applies the *decentralized* structure of running *Gibber* machines on multiple web browsers; each machine shares code text with other machines but has its own state. For example, clocks of connected machines are not synchronized and two different machines can play different music based on how the live coders choose to execute the code shared.

It seems that collaboration between live coders over a network is a natural choice. Many environments offer shared sonic environment at many levels (code text, sound, function, variable, clock, etc.) as well as communication channels such as live chat. There remain interesting questions in how live coding environments and languages can further facilitate such collaboration.

2.3 Collaboration with Instrument Performers

In the field of NIME, it is not difficult to find an ensemble where instrument performer(s) collaborates with laptop musicians in various forms (e.g., live patching, laptop orchestra). There can be many types of collaborative scenarios between live coders and non-live coding musicians

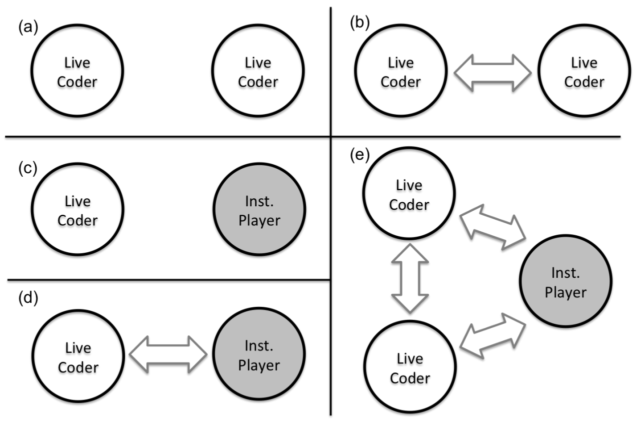


Figure 1: Possible Scenarios for Collaborative Live Coding. An arrow between two nodes means that two musicians are connected over network. Note that these are just a few examples from many possibilities.

(For example, Figure 1-(c),(d),(e)), we had difficulty finding, however, examples where live coding musicians and instrument performers play in a mixed ensemble. Perhaps, it is a rare case because of the fundamental difference in control and mapping between live coding and playing musical instruments. The algorithmic control of sound in live coding imposes multi-layered, gradually evolving music in musical aesthetic. The inclination behind this aesthetic comes from the mapping between physical control and sound in live coding. Although, an algorithm can play a multi-layered, possibly infinite number of onset events in generative music, there is a certain delay between the time one starts the gesture (programming/typing) and the time the change is finally made in music (execution of new code), we often refer to this as a *lack of immediacy.* This delay makes a live coder want to loop a musical pattern in the background so as to minimize silence during a performance. In contrast, in playing a musical instrument, a performer's gesture generates onset immediately and there is, in general, no delay between the gesture and sound. On the other hand, acoustic instrument players would struggle to play multi-layers of sound similar to what a live coder can easily achieve. We believe there will be a definite contrast when live coding music is played with instrumental music.

We anticipate this collision of making music to lead, in the near future, to the creation of unique computer music performances. Nevertheless, another approach could circumvent the challenge by forming a *sequential* relationship between live coders and instrument performers. Or live coding musicians can mediate the sonic outcome of an acoustic instrument, either by providing a musical medium for instrument performers to perform with or by processing an acoustic outcome of musical instruments with algorithms (note that the order of sequence differs for the two cases). The authors are interested in the former, a *sequential* relationship where a live coder mediates the musical expression of an instrumental musician in certain ways.

Recently, the author participated in developing an extension of *LOLC*, *SGLC* [10]. In this extended environment, laptop musicians generated real-time music notation on the fly by typing commands in the environment and instrumental musicians sight-read the generated score for collaborative improvisation. Here the outcome of text-based interaction is real-time notation, not music. The generated music score is rendered in various forms so that it gives space for instrument performers to interpret (e.g., open-form score,

[1]`http://gibber.mat.ucsb.edu/`

such as graphical or textual score). The user study showed that the system effectively integrates acoustic instrument players into a mixed ensemble. We believe the idea of mediation can be easily transformed to a collaborative live-coding scenario. The outcome of live coding need not be generative music; it can be a medium to instrumental musicians.

A previous work of the authors follows a similar mode of *sequential* collaboration between a live coder and an instrument performer [9]. In this scenario, the outcome of live coding is a musical instrument on a mobile device and the instrument performer plays the live-coded mobile music instrument. In this mode of performance, a live coder by herself cannot generate sound; an additional instrument performer is needed to do so. The sound is coming from the musical instrument built on the fly (see Figure 1-(d) and (e)). The sonic result of the performance is drastically different from traditional live coding music. While the instrument musician can only play one layer of sound with the instrument, the outcome of improvisation can demonstrate the immediate expressivity of having a musical instrument. Indeed, this would be challenging to reproduce in the traditional mode of live coding.

In our aforementioned paper, we also explored design challenges coming from the interdependent relationship between a live coder and instrument builder and an instrument performer. We suggested a set of programming techniques for live coders to coordinate smooth collaboration with the musicians who use the dynamically changing musical instrument. In the improvisation piece performed, there was also a sequential collaboration in the opposite direction from the instrument performer to live coders as well. A live coder captured a snippet of the expressive play of the instrumental musician, took trace of interaction data of the motif, and used it to create a pattern running in the background. The performance showed a wide spectrum of collaboration available in collaborative live coding scenario.[2]

3. PROGRAMMING ENVIRONMENT FOR COLLABORATIVE LIVE CODING

The question of how to make a live coding environment facilitate this mode of networked collaboration is what gave rise to the goal of this work. Following our previous work reported in [9], we utilize the programming environment for mobile music applications *UrMus* [5]. The code editor of *UrMus* is implemented as a web application running on a mobile phone so that a user can code on any web browser (usually running on a laptop) and transmit code over a local wireless network to be interpreted on the device. In this remote programming environment, multiple coders can implement a mobile music instrument together on the fly. An example of two live coders and one instrument performer is depicted in Figure 1-(e).

Through a course of rehearsals and the performance from the previous work, we realized a number of improvements remained to be made to the environment so as to support collaboration and communication in this setting. We developed an extended programming environment designed for this specific mode of performance. The design goal of the extension is to improve *UrMus* so that it will support collaborative improvisation and communication for this setting. The remainder of this paper describes a concrete extension

[2]A video of a rehearsal for the concert is available at `http://youtu.be/B9VYA_6spoI`, which was selected as the 3rd prize in live coding demo contest in LIVE workshop, ICSE 2013. Additionally the shortened video footage of our actual performance is available at `http://youtu.be/r_BGC4Wsm6c`.

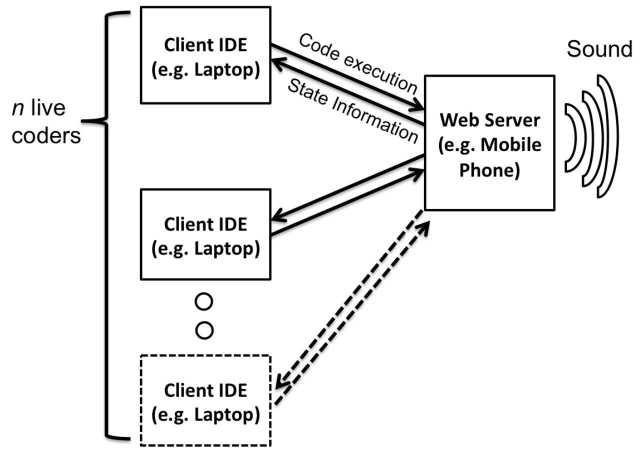


Figure 2: Centralized Architecture of Collaborative Live Coding

to the programming environment. We first introduce the overall structure of the network, go over new features added to the environment and explain the design choices that were made.

3.1 Centralized State Space

Wang introduced two models of collaborative audio programming space-server-client and peer-to-peer [14]. These terms are also related to the classification of network music in *centralized* and *decentralized* approaches that Weinberg suggested [15]. In a *centralized* approach, there is only one program state space that generates the outcome of live coding. Live coding musicians connect to the machine and remotely execute code over the network. Since only one state space is running on one machine, no additional process is needed to either share information or synchronize clocks. Whereas in a *decentralized* approach, there exist multiple machines used by multiple live coders. In this case, each machine has its own state and generates an individual outcome. Therefore it requires a separate mechanism to share information between live coders (e.g., additional server application or distributed memory).

UrMus inherently supports a *centralized* model. Live coders on laptops (multiple clients) can connect to the mobile device (server) and transmit code text to the device wirelessly. Figure 2 demonstrates the system architecture of this work. *UrMus* is based on *lua* [7], a light-weight interpreter language so that whenever code is run, the code will be evaluated and executed on top of the current state space thus far accumulated (see Figure 3-(a).) For example, if a live coder submitted code that assigned a value to a string variable named *str*, any live coder who submits code afterwards will have access to *str*. Note that although the editors is web-application, this is different from the environment of *Gibber*. Here the web-editor on the laptop is a dumb terminal that holds code text and the mobile device is the machine that synthesizes sound and renders a graphical user interface.

3.2 Individual and Shared Namespace

In the last example from the previous section, we can already find a problem when multiple programmers code in a centralized state space. What if someone else creates a variable named *str* without realizing a variable with such a name already exists? See another example in Figure 3-(a). For the collaborative live coding environment with a *centralized* approach, it is desirable to control this risk, otherwise a collaborator may accidentally overwrite the state

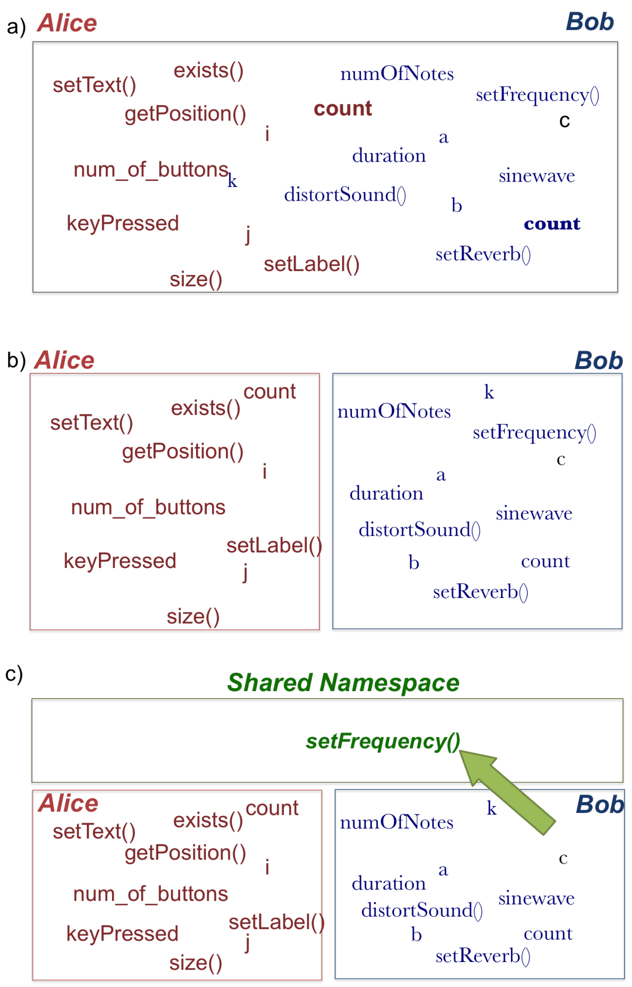


Figure 3: Example of possible namespace control. a) There is only one state space leaving open the risk you inadvertently modify someone else's state space. See there are two *count* variables, the later produced of which will overwrite the earlier one. b) Having a separate namespace per each live coder will prevent this collision but then how will they collaborate without shared state? c) There is a shared namespace that all live coders have access to. A live coder can share either a variable or function to be shared with other live coders. For example, Bob can choose to share a function "setFrequency" so that Alice can execute that function.

that someone created. One non-technological solution to this problem is live coders continually paying attention to other peoples' code text and being careful not to make conflicts as a part of live coding practice. This introduces extra cognitive loads for live coders and limits the improvisational aspects of live coding. Another naive solution to this is to make each live coder have individual state space which will prevent conflicts (see Figure 3-(b)). However, this solution obstructs basic ways to collaborate by isolating a live coder from the centralized state space. For instance, a live coder in an independent state space cannot perform critical network music gestures such as *Borrowing and Stealing* [3].

Our solution to this issue is to give each individual live coder his/her own *namespace* and to create a *shared namespace* separately that everyone has access to. Each live coder can selectively transfer any program state (e.g., variables, functions) of his/her own to the shared space (see Figure 3-(c)). In this way, live coders need not worry about someone else corrupting their code by mistake. Note that this approach requires additional user input to select a set of state to be shared, hence facilitating communication between coders. For example, borrowing is possible but stealing is not without user's permission; one has to ask the owner to share a certain state. This is contrary to the alternative where information is open to anyone but the environment only alerts a live coder whenever there is a collision. We chose the more controlled option of having a coder make explicit decision on which states to share rather than react to system alerts. However, we recognize the other option also has its own strength in its open structure.

For implementation, we utilize *lua*'s functionality of specifying a namespace (or environment in *lua* term) with a function called *setfenv()*[3]. Whenever a live coder connects to the mobile device, the mobile device assigns a unique namespace for each live coder. Therefore, whatever code executed by a live coder will change the state within the namespace that is associated with the live coder. We modified *urMus* to search the shared namespace in case that the code could not be evaluated within an individual namespace so that each live coder has access to his/her own namespace and the shared namespace.

3.3 Live Variable View

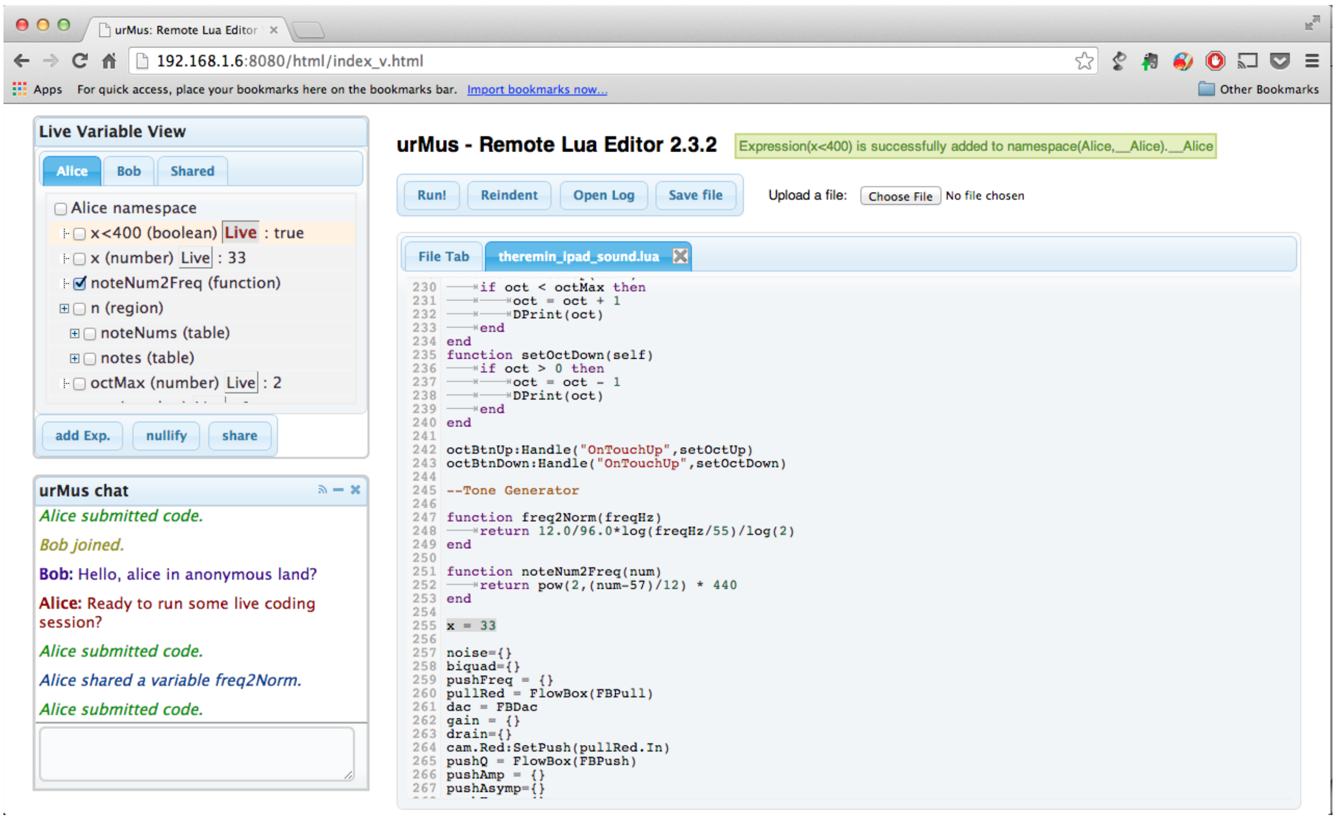
Live variable view is similar to other programming environments (e.g., Eclipse) that show you a list of variables and functions except that it shows the value of each variable at the moment. It is also similar to the debugging mode in a programming environment where you can see the state of the program at a certain point, except that you do not need to stop at a breakpoint. The live variable view displays variables, functions available in the state space at the moment, and expressions a programmer is interested in. In Figure 4, the live variable view is located in the top left corner of the editor. If a variable is abstract data typed (e.g., array, urMus GUI widgets), it shows all the elements inside the container in a hierarchical tree structure. An expression can be any code text that can be interpreted and returns a value such that you are able to evaluate any information you are interested in during the live coding session.

The state space in live coding can change over time for many reasons, including code execution on-the-fly, time-varying variables such as audio data, and user interaction influencing the program (such as pressing a button). The view updates, by default, any value changed by code execution. The *live* button right next to each entry can be used if a programmer wants to see the value of the associated entry in real time. For example, if there is a variable that contains audio sample data, you can see the actual sample value in real time.

As stated in the previous section, there are as many individual namespaces as there are live coders plus the shared namespace. Live variable view will have each namespace presented in each tab. Figure 4 shows three tabs are available in the view. Also handled in the live variable view are sharing variables, functions or expressions. One must simply check the checkbox in front of an entry that one wants to share and press the share button at the bottom. Once shared, the entry will appear in the shared namespace tab and any live coder will have access to any entry in the shared namespace.

Having live update of variables (or expressions) in live coding helps communication in two aspects. First, it provides a live view of the program state constructed and shared by each individual. The benefit of this is that it provides

[3]http://www.lua.org/pil/14.3.html



Expression(x<400) is successfully added to namespace(Alice,__Alice).__Alice

Alice namespace
- x<400 (boolean) Live : true
- x (number) Live : 33
- [✓] noteNum2Freq (function)
- n (region)
  - noteNums (table)
  - notes (table)
- octMax (number) Live : 2

Alice submitted code.

Bob joined.

Bob: Hello, alice in anonymous land?

Alice: Ready to run some live coding session?

Alice submitted code.

Alice shared a variable freq2Norm.

Alice submitted code.

```
    if oct < octMax then
        oct = oct + 1
        DPrint(oct)
    end
end
function setOctDown(self)
    if oct > 0 then
        oct = oct - 1
        DPrint(oct)
    end
end

octBtnUp:Handle("OnTouchUp",setOctUp)
octBtnDown:Handle("OnTouchUp",setOctDown)

--Tone Generator

function freq2Norm(freqHz)
    return 12.0/96.0*log(freqHz/55)/log(2)
end

function noteNum2Freq(num)
    return pow(2,(num-57)/12) * 440
end

x = 33

noise={}
biquad={}
pushFreq = {}
pullRed = FlowBox(FBPull)
dac = FBDac
gain = {}
drain={}
cam.Red:SetPush(pullRed.In)
pushQ = FlowBox(FBPush)
pushAmp = {}
pushAsymp={}
```

Figure 4: Screen capture of new urMus programming environment on a web browser. Live Variable View (topleft), Live Chat (bottomleft), code editor (right).

a summarized viewport of the program's running state and supplements code text sharing, which does not easily reflect the current state. We plan to utilize the live variable view as a gateway to share code text hierarchically so that one can re-use and modify other live coders' code.

Furthermore, live variable view enables additional communication with the instrument performer on the mobile phone. As we suggested in [9], it is important for live coders to have visual feedback from the instrument performer so that live coders understand the play of the performer, which might affect a live coder's decision on how to shape the musical instrument. Live variable view can offer a flexible monitoring tools of user interaction. For example, one can create a simple function that returns a type of musical gesture that the instrument performer is involved in and add the function as an expression (e.g., clicked, dragged, waved, etc.)

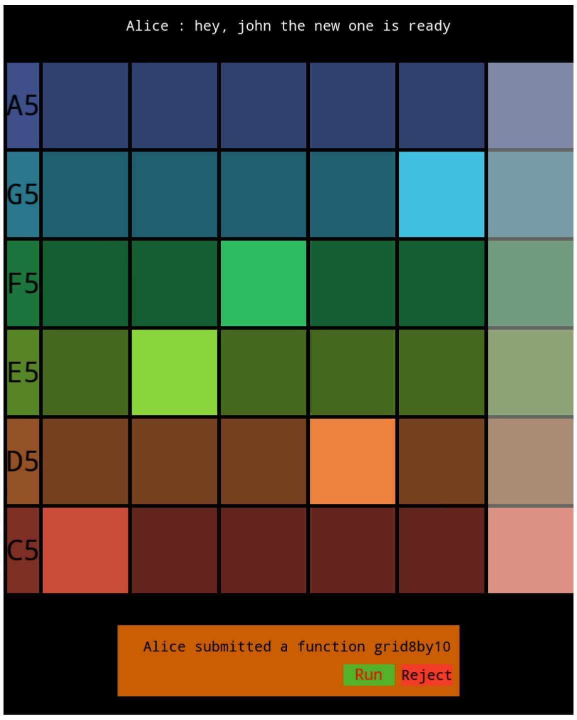
Implementation of live variable view is a composite result of web technologies such javascript/jQuery/AJAX and modification on *urMus lua* API. It keeps track of all variables and functions declared using *lua* meta-method and meta-table.[4] At the time of code execution, the live variable view makes a request to retrieve a list of variables that are newly created so that live variable view keeps the list of all the variables in a namespace. Immediately after updating the list of variables/functions, the web editor iterates all the entries in the view and retrieves the value of each variable (or expression) from the corresponding namespace. We added a functionality for the *urMus* application to transmit any data in XML format to the web editor of client. Optionally, all the entries with *live* button pressed will be updated continuously; otherwise values are only updated per code execution. The entries with the live button pressed will be updated with continuous polling from the state space. We purposely design live update of variables/expression as an option (like a check-box) so that it will minimize the network traffic and unnecessary performance degradation in a mobile device by updating all variables/expressions listed.

[4]http://www.lua.org/pil/13.html

3.4 Live Chat and Distributed Code Execution

Utilizing live chat is a common way to support communication between live coders [9, 6, 11, 14]. In this work, we improve chat support in the live coding environment by integrating numerous aspects of collaborative programming and by making chat windows informative similar to what information one obtains from a console in a programming environment. Through a live chat window, live coders will be notified when someone executes code, shares a variable, or rejects a code execution. (Rejection is discussed below.) All the chat and log messages appear on the mobile device as well, so that the instrument performer is included in the communication loop and receives the notification.

In addition, live chat enables code execution triggered by the networked instrument performer. In other words, any code typed into the chat window will not be executed immediately but will create a message window on a mobile device so that the networked performer can instead execute the code. As the system delegates the networked performer to execute the code, the performer can decide when to pull the change; otherwise, a code change at an arbitrary moment could interfere with current interaction on the instrument. The message window will state the name of the submitted function and display two buttons - run/reject. The instrument performer simply presses one of the two buttons and runs (or rejects) the newly submitted code (see the message window at the bottom of the screen in Figure 5). With this distributed code execution, the instrument performer


Alice : hey, john the new one is ready

Alice submitted a function grid8by10

Figure 5: Screenshot of an example application (tone matrix) on a mobile device. The chat message appears at the top of the screen. The remote code execution message window is presented at the bottom.

can participate in the coding process by deciding when to execute and having the option to reject. Note that the decision will be fed back to the live coders and will influence forthcoming changes. For a live coder to distribute code execution to another live coder is easy. One can write code into a subroutine function and simply share the function with others. The other live coder can then run the function with extra code. This will be useful when one wants to synchronize execution of more than one person's code.

4. CONCLUSIONS

In this work, we have introduced an extension of the programming environment to facilitate collaborative live coding, especially when musicians are connected over network. The new features have been added to support communication between live coders and a networked performer. The system provides individual and shared namespace and offers ways to distribute code execution to collaborators.

There are numerous possible directions for future work. In particular we are interested in exploring structural support for collaborative live coding by interpreting the process as a live performative co-design problem. Augmenting the chat with quantitative methods that help evaluate design decisions may help accelerate the design process, a useful proposition in a live setting. Our current system already collects design decisions by allowing an instrument performer to accept or reject code segments for execution. Information from a course of performances could be used to classify successful code sequences via machine-learning methods. From this one can create a dynamic library that offers fast lookup suggestions of code that can be rapidly inserted into an ongoing performance.

Furthermore we are interested in scaling up the current work. One can imagine that networked live coding could become a crowd-scale performance. How can collaborative coding be structured when one has to expect hundreds or more of participants to operate jointly?

Finally, coding as performance is a temporal progress. Support for navigation across a time dimension could offer increased control and interactivity. In particular we are interested in temporal navigation of the state-space along the lines of undo/redo functions, further expanding this concept towards general code notation and state visualization.

5. REFERENCES

[1] Á. Barbosa. Displaced soundscapes: A survey of network systems for music and sonic art creation. *Leonardo Music Journal*, 13:53–59, 2003.

[2] A. R. Brown and A. C. Sorensen. aa-cell in practice: An approach to musical live coding. In *Proceedings of the International Computer Music Conference*, pages 292–299. International Computer Music Association, 2007.

[3] C. Brown and J. Bischoff. Indigenous to the net: early network music bands in the san francisco bay area. *Available at crossfade. walkerart. org/brownbischoff*, 2002.

[4] N. Collins, A. McLean, J. Rohrhuber, and A. Ward. Live coding in laptop performance. *Organised Sound*, 8(03):321–330, 2003.

[5] G. Essl. UrMus – An Environment for Mobile Instrument Design and Performance. In *Proceedings of the International Computer Music Conference (ICMC)*, Stony Brooks/New York, June 1-5 2010.

[6] J. Freeman and A. Troyer. Collaborative textual improvisation in a laptop ensemble. *Computer Music Journal*, 35(2):8–21, 2011.

[7] R. Ierusalimschy. *Programming in lua*. Roberto Ierusalimschy, 2006.

[8] M. Kaltenbrunner, S. Jorda, G. Geiger, and M. Alonso. The reactable*: A collaborative musical instrument. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2006. WETICE'06. 15th IEEE International Workshops on*, pages 406–411. IEEE, 2006.

[9] S. W. Lee and G. Essl. Live coding the mobile music instrument. In *Proceedings of New Interfaces for Musical Expression (NIME)*, Daejeon, South Korea, 2013.

[10] S. W. Lee and J. Freeman. Real-time music notation in mixed laptop–acoustic ensembles. *Computer Music Journal*, 37(4):24–36, 2013.

[11] C. Roberts and J. Kuchera-Morin. Gibber: Live coding audio in the browser. In *Proceedings of the International Computer Music Conference (ICMC)*, Ljubljana, Slovenia, 2012.

[12] J. Rohrhuber, A. de Campo, R. Wieser, J.-K. van Kampen, E. Ho, and H. Hölzl. Purloined letters and distributed persons. In *Music in the Global Village Conference (Budapest)*, 2007.

[13] A. C. Sorensen. A distributed memory for networked livecoding performance. In *Proceedings of the International Computer Music Conference*, pages 530–533, 2010.

[14] G. Wang, A. Misra, P. Davidson, and P. R. Cook. Coaudicle: A collaborative audio programming space. In *In Proceedings of the International Computer Music Conference*. Citeseer, 2005.

[15] G. Weinberg. Interconnected musical networks: Toward a theoretical framework. *Computer Music Journal*, 29(2):23–39, 2005.