CodeBank: Exploring public and private working environments in collaborative live coding performance

Ryan Kirkbride
University of Leeds
ryan@foxdot.org

ABSTRACT

This paper introduces the collaborative live coding platform *CodeBank*, which utilises public and private working within musical performance over a network. *CodeBank* allows live coders to create polished live performances of improvised and experimental music using the live coding environment, *FoxDot*. The system is split into two applications; client and server. The server application generates audio for the audience while performers use the client to listen to, and experiment with, a local version synchronised with the server through headphones. Taking inspiration from collaborative software development tools, which provide version control, such as Git and Mercurial, performances in *CodeBank* involve 'pulling' codelets from the server-side central repository to listen to changes before 'pushing' the snippet back to the server.

1 Introduction

CodeBank is a performance system that offers users a private workspace to experiment in before outputting their work to a public performance space for an audience and also their co-performers. This allows performers to take larger risks with the code within their private workspace without fear of disrupting the flow of the collaborative performance taking place publicly. Each connected user is synchronised to a dedicated 'performance server' that generates audio for an audience. Changes made to code in a user's private workspace can be heard through headphones without affecting the music that the audience hears. Once a user is satisfied with the musical changes they have made, they can push their new snippet of code, called a 'codelet', to the public server. *CodeBank* implements a technique for managing collaborative projects in software development called version control in the microcosm of a live coding performance. Version control is the process of managing changes in data where developers contribute to a shared repository by pushing changes from their own private version and pulling the changes made by other contributors to keep up to date.

1.1 Motivation

This work follows on from previous research into collaborative live coding systems from which the cooperative live coding tool, *Troop*, was developed (Kirkbride 2017). Troop is a real-time concurrent live coding text editor that allows multiple users to work on the same body of code simultaneously. This is very much a public working environment as all connected users, and any audience members, can see and hear the changes made to the code immediately. Troop has been in continual development and testing through practice with The Yorkshire Programming Ensemble, made up of Lucy Cheesman, Laurie Johnson, and myself. In early sessions with the system, the process was described as "chaotic" and the sound as a "cacophony". This was in part due to the number of bugs in the software and the short amount of time we had been playing together as a group. Even in performances and practices now, nearly two years on, the combination of several different threads of musical experimentation can often lead to a harsh juxtaposition in the music, which is rarely the desired outcome.

Mistakes can also be made during performance; stopping the wrong sound or adding too much distortion, for example. Can some of the human error that creeps into improvised live coding be reduced by introducing a private workspace for experimenting with ideas? Would it be an innovative idea, or would it go against the philosophy of "embracing error" (Yorkshire Sound Women Network 2016), which is often seen as central to the practice of live coding?

The complexity of the Troop software, which utilises a text consistency algorithm called Operational Transformation (Ellis and Gibbs 1989), was another reason to pursue a different approach to collaboration in live coding. We still find that unexpected and never-before-seen errors arise during live performances with Troop that derive from the complex nature of the implementation of Operational Transformation. Could a more simple collaborative environment for live coding exist with a more reliable implementation?

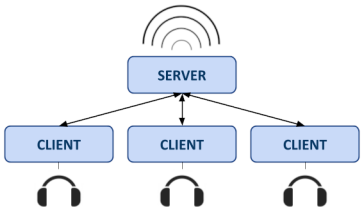

Figure 1: *Network topology of the CodeBank application with three connected users*

Already we see several research questions emerging from the reflections on practice with Troop regarding the simplicity of both its practical use and its technical requirements for development. This has prompted me to develop *CodeBank* as a collaborative live coding tool that is simple in its design and its use, but also gives performers a method for managing the multiple, and sometimes conflicting, musical ideas that occur simultaneously through the use of private workspaces.

1.2 Related work

As the popularity of live coding has increased over time, so too has the desire to create music together with other live coding musicians. The number of collaborative environments that have appeared in recent years is a testament to this. In many instances the ability to collaborate is made available to users through the live coding language itself, such as TidalCycles (McLean 2014) and Impromptu (Sorensen 2010). TidalCycles keeps live coders tightly synchronised as long as the computers used are synchronised to the same clock using standard protocols and Impromptu uses a "bulletin-board" system that allows connected users to share data easily over a network during a performance. There are also extension classes written for existing languages for collaborating over a network such as the BenoitLib (Borgeat 2010) for the SuperCollider language used by popular live coding groups *Benoit and the Mandelbrots* and *Algobabez*. Browser-based environment, Gibber (Roberts and Kuchera-Morin 2012), also allows users to collaborate over the internet using a combination of instant-messaging and cross-user code-editing.

There have also been language-agnostic interfaces developed to enable high levels of collaboration between users regardless of the language being used, such as Extramuros (Ogborn et al. 2015) and, as previously mentioned, Troop. Extramuros is a single web page interface that allocates each connected user a text box and sends evaluated code to the interpreter language of choice, such as TidalCycles or SuperCollider, and allows performers to view and edit the contents of other users' text boxes at the same time. This not only gives performers the ability to work with each other's code, it also condenses what would be displayed on multiple projectors into a single screen, giving audiences a better overview of the various coders' contributions during a performance. The Troop system is a graphical user interface (GUI) that operates in a similar manner to Extramuros but all connected performers share a single text box and their contributions are differentiated by the colour of the text.

One of the earliest examples of collaborative live coding systems was the History class written in SuperCollider, which is the predecessor of the Republic system (Blackwell et al. 2014) that allows participants to write code distributed across a network, notably used by the ensemble *Powerbooks Unplugged* (Rohrhuber et al. 2007). In essence the History class allows participants connected over a network to send small chunks of code called "codelets" to one another who can then reproduce the output of the codelets on their own computer, or change it in some way and redistribute the augmented codelet across the network. Although it is one of the earliest instances of a collaborative live coding system, much of the inspiration for *CodeBank* is taken from the History class, most notably the use of codelets being sent to all connected participants during performance.

Fencott and Bryan-Kinns (2013) found that by having their own digital "space" to work in, participants enjoyed themselves more when creating music together through a digital interface. The study required participants to create music together using the same interface but with three different control parameters; c0, c1, and c2. The first parameter, c0, had all music modules audible and visible to all connected participants, the second, c1, shared no modules with other participants unless explicitly pushed to a publicly shared space, and the last, c2, allowed participants to view other participant's modules if they wanted to by opening a new tab. Participants were then asked to fill out a questionnaire regarding their experience. Most participants felt the best music was created when they had a private workspace (c0=5, c1=12, c2=8) and they also enjoyed themselves more (c0=5, c1=13, c2=8). Interestingly, participants felt they edited the music together the most when working with interfaces with private spaces (c0=4, c1=11, c2=8) but also felt they worked more on their own (c0=3, c1=10, c2=13). In their conclusion they suggest that splitting musical interaction into public and private spaces should be a "key design consideration" for collaborative musical interfaces, and is at the nexus of the *CodeBank* project.

The *CodeBank* system also takes inspiration from popular version control tools, such as Git and Mercurial, that are used in software development projects. Version control keeps a history of changes made to a shared repository of code, including

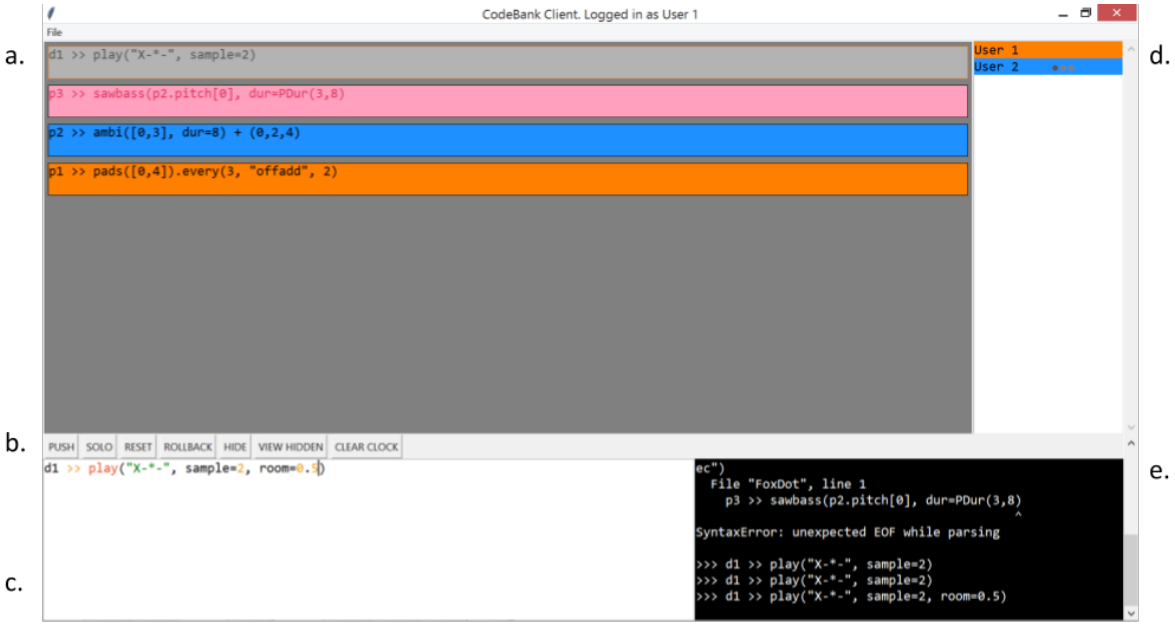

Figure 2: *A screenshot of the CodeBank client interface with features labelled A-E*

the identity of a contributor, and developers work on their own private version of the repository by pushing and pulling the changes made by themselves and their team. The parallel between private and public working in version control and the suggestions for musical interaction by Fencott and Bryan-Kinns (2013) resonated with me, considering that live coding shares traits with both software engineering and musical interaction.

2 Implementation

CodeBank is an interface to the live coding library, *FoxDot* (Kirkbride 2016), which is written in Python and creates audio by sending messages using Open Sound Control (OSC) to SuperCollider to trigger pre-written synths. To better incorporate specific features of *FoxDot* into the interface, *CodeBank* is also written in Python using the standard library's `Tkinter` GUI builder library. The source code for *CodeBank* can be found at https://www.github.com/Qirky/CodeBank. The system is split into two applications; a client and server. As shown in Figure 1, the server connects multiple clients to one another and also assumes the role for generating the audio for an audience through speakers. The client program allows participants to run code in a local environment and then push code to the server, which is run for the audience to hear. Participants using the client program can listen to the audio created in the local workspace using headphones, which is synchronised to the audio produced by the server.

2.1 Client application

Once a user opens the client application they will need to connect to a running *CodeBank* server and enter the password and a username. They will then be able to write and run code locally and push code to the server to be run for an audience to hear. Each connected user is allocated a different colour by the server. This helps both audience members and performers differentiate the contributions made by each performer. Code written by a client is pushed to the server where it is stored as a codelet. Each codelet on the server is displayed for all connected clients in the 'Public Repository' section of the interface. By clicking on one of these codelets, a user can pull it to their local workspace and begin editing the codelet, before pushing the codelet back to the server. While being edited, the codelet in the 'Public Repository' is locked and cannot be edited by another user. Once a codelet is edited and pushed back to the server, it becomes unlocked and its contents is updated.

Figure 2 shows what typical *CodeBank* session looks like with several features labelled. The 'Public Repository' (a) is the collection of codelets currently residing on the server that have been pushed by connected users. Each codelet's colour relates to the user that pushed that codelet to the server. This is done from the local workspace at the bottom of the interface. Codelets are coloured pink if they have been uploaded with a syntax error in order to highlight the problem and make it clear it needs to be fixed for the code to work. A grey codelet with a coloured outline indicates that the codelet has been locked because it has already been pulled from the server and is being edited by the user with the same colour as the outline.

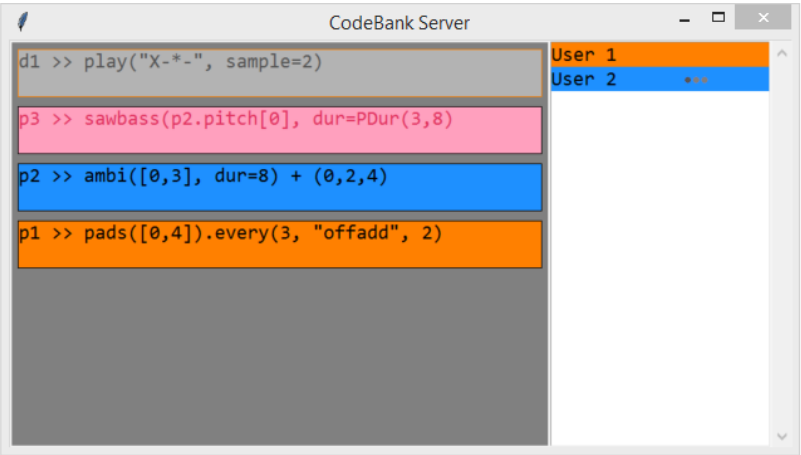

Figure 3: *A screenshot of the CodeBank server interface with several codelets running.*

Codelets with a grey background cannot be pulled by clients until they are pushed by the client that last pulled it so as to avoid multiple users editing the same codelet and overwriting each other's changes.

The 'Action Buttons' (b) are a number of buttons for specific actions relating to the current codelet in the local workspace. These actions are `PUSH`, `SOLO`, `RESET`, `ROLLBACK`, `HIDE`, `VIEW HIDDEN`, and `CLEAR CLOCK`. The first action button, `PUSH`, sends the contents of the local workspace to the server. If the user had pulled a codelet from the server, it is updated with the new codelet, else it will create a new codelet on the server. The `SOLO` action button will use the *FoxDot* `solo()` function to isolate the sound produced by the codelet in the local workspace so that a user can hear the layer more clearly. Pressing the `SOLO` button again will undo this effect and reintroduce the sounds generated by other running codelets. Using the `RESET` button will undo any local changes made to a codelet in the local workspace and allow the codelet to be edited by other users. A codelet can be reverted to a previous version by using the `ROLLBACK` action button after pulling it from the public repository. If a codelet is no longer needed, it can be removed, but not deleted, from the public repository using the `HIDE` action button. Hidden codelets can be shown to the local user by pressing the `VIEW HIDDEN` action button. A hidden codelet can be re-introduced by using the `VIEW HIDDEN` button, clicking on a hidden codelet to pull it from the server, then using the `PUSH` action button. The `CLEAR CLOCK` action button is used to stop all the sound on the server.

The 'Local Workspace' (c) is a text box, which is used to create and edit codelets that are pushed to the server. A user can run the code in the local workspace on their machine only by using the keyboard shortcut, `Ctrl+Return`. This allows the user to hear the effects of their changes without affecting the sound heard by the audience. Once happy, the user can use the `PUSH` action button to send the codelet to the server and clear the text box. The 'User Directory' (d), located in the upper-right of the interface, is a list of connected users displayed with the corresponding user-colours. This can be used to quickly work out which participant last edited which codelet from looking at the colours of the matching boxes. An ellipses is shown next to a name to indicate that a user is creating a new codelet. The 'Console' (e) is displayed in the bottom-right of the interface for giving the user text feedback on code run in the local workspace and from the public repository. This helps users keep track of errors or view certain data using the Python `print` command.

2.2 Server application

The server application serves several functions; to connect participants to one another and distribute codelets, to generate audio for an audience, and to display information about the code and connected users via a projector screen. The server interface is similar to the client interface but only displays the 'Public Repository' and 'User Directory' components. Code pushed to the server from clients is stored as a codelet object on the server with a history of changes made to it. These versions can be accessed by clients by using the `ROLLBACK` action button once a codelet has been pulled from the server.

The server shares several similarities with the client interface; codelets are displayed in the colour of the last user to edit them and are also greyed out when locked for editing and changed pink when uploaded to the server with a syntax error. The server also displays an ellipses next a user's name when they are creating a new codelet. This gives members of the audience access to some of the process of a live coding performance, including the deobfuscation of error. What audiences don't see, however, is the indiviual characters being added or deleted as they would in a traditional live coding performance. While this may go against the much-quoted TOPLAP manifesto line, "Obscurantism is dangerous. Show us your screens." (TOPLAP 2004), the change of the code over time is still shown throughout a performance.

Figure 4: *A photo of an early CodeBank practice session*

3 Conclusions and further work

The development of *CodeBank* is still in its early stages and, at the time of writing, it has only been used a small number of times in practice with The Yorkshire Programming Ensemble. The *CodeBank* server was set up on a Raspberry Pi 3 connected to a screen and a set of speakers, which users connected to over a local wireless network, as shown in Figure 4. From the first rehearsal using *CodeBank* all performers noted that the style of live coding differed considerably compared to previous collaborative performances using Troop. One user stated:

> "It changed the way I was interacting with the code in that I was being more thoughtful about the changes I was making, but consequently paying less attention to what you guys were doing. Compared to using Troop where I have a general awareness of what you're both up to. I think it slowed me down a bit but also encouraged more significant changes rather than incremental ones."

The slower process of coding meant that each time code was added or changed, it had more impact on the overall sound but these changes did not occur very frequently. Often large portions of time went by when every user was editing their own local version of code then pushing their changes to the public repository simultaneously, resulting in large shifts in the music. While this was an exciting process to be part of, it was also quite uncomfortable because the music would tend to change in a way you would not expect it to. Being able to experiment in your own workspace meant that any incremental change made to the code was only heard by the local user and would not give any indication to anyone else as to where the sound was headed. With time and practice, however, techniques could be developed to better coordinate code development in performance, and it will be interesting to compare them with those developed while using the Troop software.

It also emerged that there was a need for improving the user experience of the system, such as adding more control options from the keyboard. Currently, the action buttons, such as `PUSH`, can only be activated by clicking them with the button but live coders tend to use keyboard shortcut commands to control their interfaces and they felt that using the mouse disrupted the flow of the session. It was suggested that a keyboard shortcut with three keys could be used, such as `Ctrl+Shift+Enter`, to push code to the public repository to make it harder to accidentally do so while running code in the local workspace using the keyboard shortcut, `Ctrl+Return`. Furthermore, selecting codelets from the public repository is also done through mouse clicks and all performers felt that this process could be simplified through keyboard control.

Another possible extension to the *CodeBank* system would be to make it language agnostic in the same vein as Extramuros and Troop to allow for a wider range of live coders to engage with this style of collaboration. Troop is a GUI written in Python that interfaces with other interpreted languages, such as TidalCycles and Sonic-Pi, and it would be feasible that *CodeBank* could use similar methods to communicate with other live coding environments. However, *CodeBank*'s action buttons are specifically tied to features in *FoxDot*, such as `SOLO` and `RESET`, which would mean these features would also have to be reproducible in the desired host language.

CodeBank is designed to help reduce human error and improve the overall quality of collaborative live coding performance but there is still some way to go before its success can be evaluated. There is, however, something to be said about these design goals in the context of the philosophy of live coding. Live coding is often described as "embracing error" and letting

failure lead you in musical performance (McLean 2017), but *CodeBank* arguably does the opposite. While it does provide a safety net for experimentation, it also lets users try and fail with ideas without fear of doing so in front of a live audience. The *CodeBank* system actively encourages users to experiment and let error and failure guide them in performance but in a space they can be comfortable in doing so. Unfortunately this does lead to a delay between the formation of ideas and their eventual sonification for the audience. It could be argued that this reduces some of the "liveness" in live coding, but the counter argument is that by allowing performers to experiment in a local workspace, *CodeBank* supports improvisation and the creation of spontaneous musical ideas.

3.1 Acknowledgments

I would like to thank the White Rose College of Arts and Humanities (WRoCAH) for funding my research and my supervisors, Dr. Guy Brown and Dr. Luke Windsor, for helping me develop the idea for *CodeBank*. I would also like to thank my good friends, Lucy and Laurie, for testing out my ideas and making weird music with me.

References

Blackwell, Alan, Alex McLean, James Noble, and Julian Rohrhuber. 2014. "Collaboration and Learning Through Live Coding (Dagstuhl Seminar 13382)." *Dagstuhl Reports* 3 (9). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Borgeat, Patrick. 2010. "Cappelnord/Benoitlib: SuperCollider Extensions Used by Benoît and the Mandelbrots." https://github.com/cappelnord/BenoitLib.

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

Fencott, Robin, and Nick Bryan-Kinns. 2013. "Computer Musicking: HCI, Cscw and Collaborative Digital Musical Interaction." In *Music and Human-Computer Interaction*, 189–205. Springer.

Kirkbride, Ryan. 2016. "FoxDot: Live Coding with Python and Supercollider." In *Proceedings of the International Conference of Live Interfaces*, 194–98.

———. 2017. "Troop: A Collaborative Tool for Live Coding." In *Proceedings of the 14th Sound and Music Computing Conference*, 104–9.

McLean, Alex. 2014. "Making Programming Languages to Dance to: Live Coding with Tidal." In *Proceedings of the 2nd Acm Sigplan International Workshop on Functional Art, Music, Modelling & Design*, 63–70. ACM.

———. 2017. "Live Coding – Potac – Medium." https://medium.com/potac/live-coding-1eb06f0ddf26.

Ogborn, David, Eldad Tsabary, Ian Jarvis, Alexandra Cárdenas, and Alex McLean. 2015. "Extramuros: Making Music in a Browser-Based, Language-Neutral Collaborative Live Coding Environment." In *Proceedings of the First International Conference on Live Coding*, 163–69.

Roberts, Charlie, and JoAnn Kuchera-Morin. 2012. *Gibber: Live Coding Audio in the Browser*. Ann Arbor, MI: Michigan Publishing, University of Michigan Library.

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*, 530–33.

TOPLAP. 2004. "ManifestoDraft - Toplap." http://toplap.org/wiki/ManifestoDraft.

Yorkshire Sound Women Network. 2016. "Live Coding - Youtube." https://www.youtube.com/watch?v=PboSZGllzsU.