TROOP: A COLLABORATIVE TOOL FOR LIVE CODING

Ryan Kirkbride
University of Leeds
sc10rpk@leeds.ac.uk

ABSTRACT

Live Coding is a movement in electronic audiovisual performance that emerged at the turn of the millennia [1] and is now being performed all over the world through range of artistic practices [2]. It involves the continual process of constructing and reconstructing a computer program and, given that most performances are improvisational in nature [3], working together from multiple computers can provide a challenge when performing as an ensemble. When performing as a group Live Coders will often share network resources, such as a tempo clock to coordinate rhythmic information [4, 5], but rarely will they work together directly with the same material.

This paper presents the novel collaborative editing tool, Troop, that allows users to simultaneously work together on the same piece of code from multiple machines. Troop is not a Live Coding language but an environment that enables higher levels of communication within an existing language. Although written in Python to be used with the Live Coding language FoxDot [6] Troop provides an interface to other Live Coding languages, such as SuperCollider [7], and can be extended to include others. This paper outlines the motivations behind the development of Troop before discussing its applications as a performance tool and concludes with a discussion about the potential benefits of the software in a pedagogical setting.

1. INTRODUCTION

1.1 Collaborative Live Coding Environments

Rather than simply sharing the same clock when playing as a group Live Coders may choose to use a program that specifically facilitates a collaborative or synchronised performance. Such tools tend not to be programming languages themselves but software designed to aid communication between multiple computers using an existing Live Coding language. A useful example of this is the browser-based system Extramuros [8], which allocates each connected performer a small text box on a web page into which they can each write code. These text boxes are visible and can be edited by any other connected performer. Extramuros is "language-neutral" and can be used with any language that allows commands to be "piped" into it, which improves accessibility to a wider range of Live Coding practitioners.

Another browser-based tool for collaborative Live Coding is Gabber [9]; an extension to the Gibber library [10] that combines a chat-room interface and shared text buffers similar to Extramuros. Unlike Extramuros the edited code in each users' text buffers is only executed on that user's machine. Rohruber et al. [11] use an interface within SuperCollider similar to that of a chat-room to share small blocks of code dubbed "codelets". In contrast to Extramuros and Gabber, the "codelets" are shared with, but not executed on, each connected machine. Performers can choose to use or modify these chunks of code and submit them to the chat-room interface. This implementation stems from the performance style of PowerBooks Unplugged; the Live Coding ensemble discussed by Rohruber et al. [11], whose members sit at laptops at various points in the room and create a varying yet connected sonic experience for audiences. Since then Rohrhuber has gone on to develop a popular SuperCollider extension called The Republic [12] that allows performers connected over a network to access and modify one another's code. Similarly, LOLC is a "textual performance environment" that aims to facilitate methods of practice common to both improvisation and composition, with a focus on conversational communication [13]. It is a platform for sharing shorthand musical patterns, which are then played or transformed and re-shared by other participants.

As opposed to sending text between performers, Impromptu Spaces uses a tuple-space that acts as a "remote bulletin-board" for posting and retrieving information across a network [14]. This creates a shared and distributed memory that is accessible to each connected client and allows users to manipulate global variables such as tempo while avoiding any read/write clashes.

1.2 Motivation

Live Coding ensembles will often play connected over a network in order to synchronise their music and share information between one another. This presents several challenges for both the performers and audience during a live performance. One of the main technical challenges facing a Live Coding ensemble is that of temporal synchronisation. Latency (the time it takes for messages to be sent between computers) in network communication has to be accounted for when aligning multiple performers' computer clocks. When performing over large geographical distances a delay is affordable as long as the synchronisation between parts occurs at each end of the connection.

This style of synchronisation is used by Ogborn [15] and his network music duo, "Very Long Cat', that synchronises Live Coding and tabla drumming over the internet.

Alternatively, two or more Live Coders might perform together at the same location in front of an audience with each performers' laptop creating sound separately. Depending on the style of music this requires each machine to be temporally synchronised in real time, which is not always easy to accomplish. A common way to do this is through a clock-sharing protocol where each machine is constantly listening to a designated time-keeper, as implemented by the Birmingham Ensemble for Electroacoustic Research [4] and Algorave pioneers, Slub [5] for example. This requires an extra layer of configuration and is still liable to lag between performers' downbeat depending on the network being used.

Another resulting problem of this style of performance is the effect it has on the audience. Multiple Live Coders working on individual portions of code will usually have their work on separate screens, some, or all, of which will be projected for the audience. This can result in a non-optimal audience experience as there may either be too much going on or, if there aren't enough projectors available, not enough. One possible solution to this problem is the shared-buffer networked Live Coding system, Extramuros [8]. As mentioned previously this software uses a client-server model to connect multiple performers within a single web page. This allows them to create their own code and request and modify other performer's code in the same window, reducing the number of screens necessary to project during the performance. However, as the number of connected users, and consequently text boxes, increases the font size must be reduced and the code becomes less legible.

Gabber [9], the network performance extension to the JavaScript based language, Gibber, works in a similar way but also allows users to interact with each others code directly within the same text buffer. This allows for only one screen to be projected but all performers' work to be displayed. Originally Gabber used "a single, shared code editor" but it was found to be problematic "as program fragments frequently shifted in screen location during performances and disturbed the coding of performers". Gabber has since moved to a more distributed model. The single, shared text buffer model may have proved problematic in this instance but has seen much mainstream success, most notably in Google Docs [16], and has prompted me to create a similar tool for concurrent Live Coding collaboration entitled Troop.

2. DESIGN

The purpose of the Troop software is to allow multiple Live Coders to collaborate within the same document in such a way that audience members will also be able to identify the changes made by the different performers. This section outlines the steps taken to achieve this and briefly describes the two types of network architecture used in this project.

2.1 Development Language

Troop is designed to work with the Live Coding language, FoxDot [6] but, like Extramuros, it can be language neutral with a small amount of configuration. The programming language Python [1] is useful for fast software development and comes with a built-in package for designing graphical user interfaces (GUIs) called `Tkinter`. As FoxDot is written in Python it makes it easy to pipe commands to its interpreter from Troop.

2.2 Interface

The philosophy behind Troop is that all performers seemingly share the same text buffer and contribute to its content at the same time. To allow each performer to differentiate their own contributions from others each connected performer is given a different coloured label which contains their name. This label's position within the input text box is mapped to the location of the respective performer's text cursor, as shown in Figure 1.

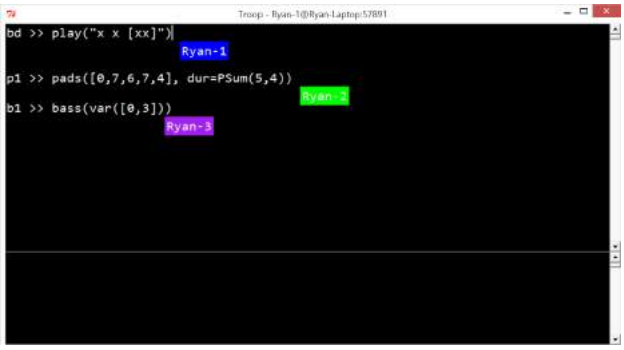

Figure 1. A screen shot of the first iteration of the Troop interface with three connected users.

As time elapses during a performance and the text buffer begins to fill up it is not always possible to separate the individual contributions and it becomes unclear as to whom has written what; even to the performers. This problem has been identified by Xambó et al. [17], stating there is a challenge in "identifying how to know what is each others code, as well as how to know who has modified it".

To help combat this problem and differentiate each performers' contributions each connected performer is allocated a different colour for text (see Figure 2) and highlighting. By doing this performers can leave traces of their own coloured code throughout the communal text buffer. Editing a block of another performer's code interweaves their colours and thought processes, creating a lasting visual testament to a collaborative process; or at least until someone else makes their own edit. The colour of the text entered by each performer matches the colour of their marker to retain a consistency in each performers' identity. It is customary in Live Coding for evaluated code to be temporarily highlighted and by doing this in separate colours it allows both audience and performers to identify the source of that action.

[1] `http://python.org/`

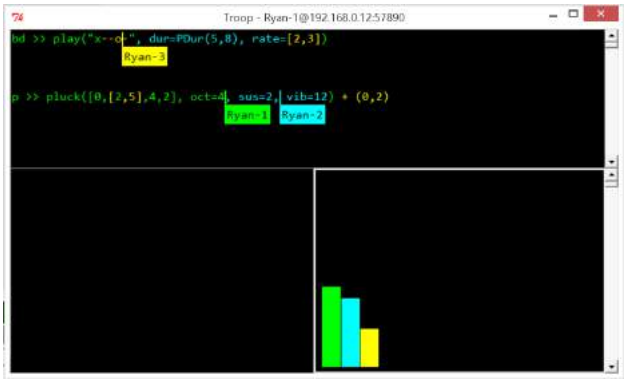

Figure 2. A screen shot of the second iteration of the Troop interface with three connected users.

Performers contributions are also measured in terms of quantity of characters present. In the bottom right corner of 2 there is a bar-chart that displays the proportion of code contributed by each performer. As a collaborative performance tool it is particularly interesting to keep track of this information and use it as a creative constraint for a performance (see section 4.2.1 for a discussion on this).

2.3 Network Architecture

The Troop software uses a client-server model but can be set up in two ways based on which machine will be handling code execution.

2.3.1 Server-Side Execution Model

This first design uses a centralised synchronous networked performance model similar to that of LOLC [13] and transfers messages using TCP/IP to ensure all messages are sent over the network. The server runs on a machine dedicated to listening to client applications (see figure 3), which are sending keystroke information, and handles the code execution and sound creation. Incoming keystroke data are stored in a queue structure to avoid sending data sent from two clients simultaneously in differing orders to other clients. As the number of connected clients increases so too would the chance of this error occurring. A disadvantage of storing all keystroke data in a queue on the server is that there is sometimes a noticeable latency between pressing a key and the corresponding letter appearing the text editor depending on the network.

In this topology only the server machine executes code and schedules musical events, which eliminates any need for clock synchronisation. This method also utilises a centralised shared name-space, which doesn't require information about the code to be sent to the client applications. The disadvantage to this approach is that all performers must be "local" to the server (a client and server can be run on the same machine together) if they want to hear the performance, unless they stream the audio through another program from the server machine.

2.3.2 Client-Side Execution Model

When performers are not co-located it might be required that code is run on the client machines as opposed to the server. In this case, the *Troop* server is run with an extra command-line argument:

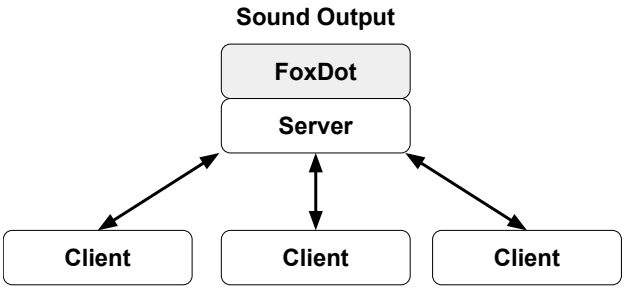

Figure 3. Network diagram for Troop's server-side execution model.

```
./run-server.py --client
```

This signals to the server to send information about what code to evaluate back to the client. Figure 4 shows how the code execution and sound creation are shifted to the client's responsibility in this set up. A client-server model was adopted over a peer-to-peer topology as it meant that the client application did not have to be reconfigured based on where the code was being executed and the centralised server queue structure for storing keystrokes could still be used.

The nature of Troop's concurrent text editing means that code in each client's text buffer is identical. This means that there is no need to use a shared name-space, such as in [14,18], as the data is reproduced on each connected machine. An advantageous consequence of this is that Troop need only send raw text across the network and no serialising of other data types is required in order for the program to be reproduced on multiple machines.

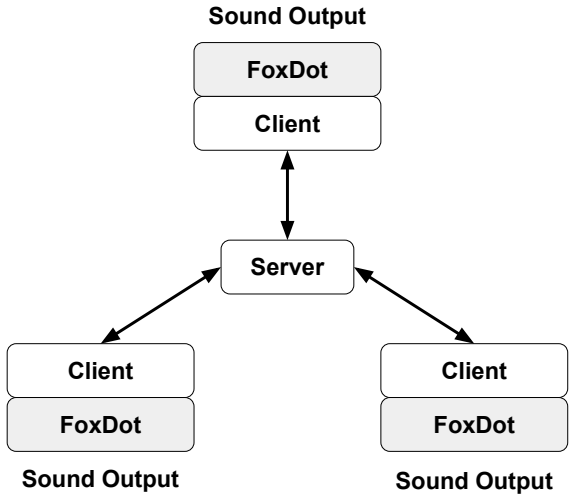

Figure 4. Network graph of *Troop*'s client-side execution model.

2.4 Extending the software for other languages

Troop contains a module called `interpreter.py` that defines the classes used for communicating with host Live Coding languages. Currently there are interpreter classes for the FoxDot and SuperCollider languages. If a language can take a raw string input and convert that to usable code then Troop will be able to communicate with it. Specifying the `lang` flag at the command line, followed by the name of the desired language, will start the client with a different interpreter to FoxDot like so:

```
./run-client.py --lang SuperCollider
```

By giving performers flexible access to multiple Live Coding languages Troop can enhance musical expressivity and broaden the channels for collaboration.

3. SYNCHRONISATION

3.1 Maintaining Text Consistency

One of the most difficult aspects to this project was making sure that the contents of the text buffer for each connected client remained identical. In Troop's first iteration, text would be added immediately on the local client then sent to the server to be forwarded to the other connected clients. Problems arose mainly when two clients were editing the same line, which caused the characters to be in different orders for each client. To combat this the clients in the next version of Troop would send all keystroke information to the server, which dispersed them to each client in the same order. This worked very well if the server application was running on the local machine but there were noticeable delays between key-press and text appearing on screen when connecting to a public server running Troop in a London data centre.

A solution was to combine these two input systems to maximise efficiency and still maintain consistency between client's text. When a key is pressed, Troop checks if there are any other client's currently editing the same line as the local client and if there are, it sends all the information to server to make sure the characters are entered in the same order for every client. If the local client is the only one editing a line the character is added immediately on the local machine and then sent to the server to tell other clients to add the character. This reduces latency for character inputs when clients are not interfering with one another but also maintains consistency at times when they are.

3.2 Limiting Random Values

When random number generators are utilised to define pitches and rhythms their values may become inconsistent between each machine and each performer will consequently hear a different piece of music. This can be combated by selecting the random number generator's seed at the start of the performance. In FoxDot for example, evaluating the command `Random.seed(1)` will do so for all connected clients and will consequently mean that any random elements will, although appear random, be completely deterministic for that session.

3.3 Clock Synchronisation

Along with information about each client's keystrokes, the Troop server sends a "ping" message to each connected client once every second. This serves two purposes; the first is to periodically check that each client can still be contacted and remove those that cannot from its address book, and the second is to aid clock synchronisation. A consequence of this "ping" message is that the `settime` method of Troop's interpreter class (Section 2.4) is called with a value, which can then communicate with whatever time-keeping data structure is used by the host language if need be.

4. CONCLUSIONS

This paper has presented the Live Coding environment, Troop, and discussed its potential as a tool for enhancing channels for collaboration and communication in Live Coding ensembles.

4.1 Applications in Teaching

Music has been used as a useful teaching analogy for computer science in younger age groups, as demonstrated by software such as Scratch [19] and the Live Coding language Sonic-Pi [20], and collaborative Live Coding can help move student thinking "from an individual to a social plane" [17].

By running several Troop servers in a classroom environment Troop would encourage collaborative work between students and allow a single teacher to monitor the work of multiple groups of students from a single machine.

4.2 Future Extensions

4.2.1 Adding Creative Constraints

Troop is an environment that facilitates group work over a network but each contributor in a document still maintains their own identity within the process through the use of coloured code. Maintaining this identity requires the number of characters entered by each user to be stored. This information could be used to add a constraint to a particular session to explore different collaborative approaches to Live Coding.

On each keystroke it would be possible to test a constraint and then only add the character to the text buffer if that test returns true. An example of this might be to disallow a single user to take up more than 50% of the text buffer, which could easily be achieved using the short function outlined in pseudo-code in Figure 5. The start of the performance would be difficult as performers would have to work together slowly to make sure they don't spend long periods of time waiting for other users to catch up, but as the total number of characters in the text buffer increased so too would the flexibility for writing larger portions of code in one go.

Another example might be to "ban" a random user from typing for a number of seconds to try and force them to look at what the rest of their ensemble are doing, or to only allow them to edit a single line of code written by one

```
def constraint(text, client):
    if client.chars > text.chars / 2:
        return False
    else:
        return True
```

Figure 5. Example pseudo-code for a creative constraint for Troop.

of their co-performers. The next iteration of Troop will implement a `constraints.py` module that will contain several creative constraints and allow users to define their own functions.

4.2.2 Web Browser Version

Live Coding environments such as Gibber [10] and LiveCodeLab [21] run in most standard web browsers and require no installation on the client machine. This lowers the barriers to entry and also makes the application platform-independent (provided the web browser itself runs correctly). Moving Troop to a Javascript based client could help provide a simpler set up procedure for those newer to computer programming.

4.3 Evaluation

Currently a small group of performers are learning to use the FoxDot and Troop software in order to perform at an event at the end of April 2017. Feedback from both the users and audience members will be collected, assessed, and contribute to the further development of Troop.

Acknowledgments

This research is funded by the White Rose College of Arts and Humanities (`http://www.wrocah.ac.uk`).

5. REFERENCES

[1] N. Collins, A. McLean, J. Rohrhuber, and A. Ward, "Live coding in laptop performance," *Organised sound*, vol. 8, no. 03, pp. 321–330, 2003.

[2] TOPLAP, "TOPLAP — the home of live coding," http://toplap.org/, 2004, accessed: 08/12/16.

[3] T. Magnusson, "Herding cats: Observing live coding in the wild," *Computer Music Journal*, vol. 38, no. 1, pp. 8–16, 2014.

[4] S. Wilson, N. Lorway, R. Coull, K. Vasilakos, and T. Moyers, "Free as in beer: Some explorations into structured improvisation using networked live-coding systems," *Computer Music Journal*, vol. 38, no. 1, pp. 54–64, 2014.

[5] A. McLean, "Reflections on live coding collaboration," *Proceedings of the Third Conference on Computation, Communication, Aesthetics and X. Universidade do Porto, Porto*, p. 213, 2015.

[6] R. Kirkbride, "FoxDot: Live coding with python and supercollider," in *Proceedings of the International Conference of Live Interfaces*, 2016, pp. 194–198.

[7] J. McCartney, "Rethinking the computer music language: Supercollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[8] D. Ogborn, "d0kt0r0/extramuros: language-neutral shared-buffer networked live coding system," https://github.com/d0kt0r0/extramuros, 2016, accessed 13/12/16.

[9] C. Roberts, K. Yerkes, D. Bazo, M. Wright, and J. Kuchera-Morin, "Sharing time and code in a browser-based live coding environment," in *Proceedings of the First International Conference on Live Coding*. Leeds, UK: ICSRiM, University of Leeds, 2015, pp. 179–185.

[10] C. Roberts and J. Kuchera-Morin, *Gibber: Live coding audio in the browser*. Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 2012.

[11] 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.

[12] A. de Campo and J. Rohrhuber, "Republic," https://github.com/supercollider-quarks/Republic, accessed: 05/02/2017.

[13] J. Freeman and A. Van Troyer, "Collaborative textual improvisation in a laptop ensemble," *Computer Music Journal*, vol. 35, no. 2, pp. 8–21, 2011.

[14] A. C. Sorensen, "A distributed memory for networked livecoding performance," in *Proceedings of the ICMC2010 International Computer Music Conference*, 2010, pp. 530–533.

[15] D. Ogborn, "Live coding together: Three potentials of collective live coding," *Journal of Music, Technology & Education*, vol. 9, no. 1, pp. 17–31, 2016.

[16] Google, "Google docs - create and edit documents online, for free," https://www.google.co.uk/docs/about/, 2017, accessed: 02/02/17.

[17] A. Xambó, J. Freeman, B. Magerko, and P. Shah, "Challenges and new directions for collaborative live coding in the classroom," 2016.

[18] S. W. Lee and G. Essl, "Models and opportunities for networked live coding," in *Live Coding and Collaboration Symposium*, vol. 1001, 2014, pp. 48 109–2121.

[19] A. Ruthmann, J. M. Heines, G. R. Greher, P. Laidler, and C. Saulters II, "Teaching computational thinking through musical live coding in scratch," in *Proceedings of the 41st ACM technical symposium on Computer science education*. ACM, 2010, pp. 351–355.

[20] S. Aaron, "Sonic pi performance in education, technology and art," *International Journal of Performance Arts and Digital Media*, vol. 12, no. 2, pp. 171–178, 2016.

[21] D. Della Casa and G. John, "Livecodelab 2.0 and its language livecodelang," in *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*. ACM, 2014, pp. 1–8.