patching music together:
collaborative live coding in pure data

IOhannes m zmölnig
Institute of Electronic Music and Acoustics
University of Music and Dramatic Arts
Graz, Austria
zmoelnig@iem.at

ABSTRACT

Live Coding has established itself as a performance practice within the field of computer music in the last few years. One of the main motivations for this technique has been to provide a level of corporality of the performance and interaction between performers and auditory, which are often lacking in traditional computer music as opposed to other music performance techniques. In this paper we present a Pd-based environment suitable for Live Coding and discuss its fitness for interactive performance.

Special focus is given to the direct interaction between multiple performers on- and off-site that exceeds the traditional playing-hearing-reacting cycle.

Keywords

Live-Coding, Collaborative Development, Pure data

1. INTRODUCTION

"Live Coding" is a relatively young performance practice that has established itself in the field of inter-media art within the last few years. By the term "Live Coding" we understand a media performance, where performers create and modify their software-based instruments in real time during the performance, as opposed to traditional computer music performances, where pre-produced content is played back ("tape music") or the musicians are performing with ready made software-instruments by starting (or scheduling) and parameterising unit-generators, effects or algorithms.

In Live Coding performances, the algorithms used to generate media content are thought of and written in real-time before the audience, possibly leading to unknown and sometimes unexpected (even to the performers themselves) results.

Like in other live musical performances, it is most fun to perform not as an individual but as a group.

2. QUESTS IN LIVE-CODING

2.1 The Quest for Code-Literacy

In order to allow the audience to follow the on-the-fly evolution of algorithms, a common practice among Live Coders is the projection of code. While this exposes the code (and thus the algorithms implemented by this code), the audience is confronted with another intellectually challenging layer: reading and understanding written (and executed) software. While this is probably not a problem for code-literate people (e.g. software developers), it is usually less satisfactory for people who are not used to a certain language (or who are not programmers at all).

This is usually amplified by the use of text-based languages that offer "elegant" but condensed solutions that are hard to grasp if one is not familiar with a certain idiom.

We believe, that using graphical programming environments like Pd[10], this challenge can be somewhat facilitated. While one can argue that graphical languages tend to express control structures in a rather cumbersome way (as compared to textual languages)[9], the simple metaphor of data-flow allows people who are not familiar with the language to at least get an overall idea what might be going on. After all, programming in data-flow languages like Pd has been directly derived from real patching (wiring) of electronic modular instruments.

For those who do not want to read the patch as algorithms at all, graphical languages can at least offer eye-pleasing, less technoid visuals.

2.2 The Quest for Collaboration

Like other collaborative improvisations, Live Coders can implement a 3-phase way of playing together: After a phase of playing (aka: coding), one enters a phase of listening to the combined result which leads to reflection on the outcome and provokes further steps.

A more elaborate way of playing together can be achieved by sharing data, e.g. by directly using the signal output of a co-player, or by sharing control data via a network.

Nevertheless all these techniques are superficial, as they only allow the players to interact on the level of symptoms of the used algorithms, rather than on the algorithmic level.

In order to overcome this limitation, one has to share the algorithms between multiple players: in the case of Live Coding this is equivalent to sharing the code in real time.

One solution for this task has been implemented in the Just-In-Time Library (JITlib)[11] for SuperCollider[5], where one player writes a snippet of code, which is then transmitted to the co-players, who can then modify the snippet and re-commit it[3]. While this allows for interaction on

the algorithmic level, the evolution is again restricted to a code-commit-recode cycle, since only valid (this is: syntactically correct) snippets can be committed and the changes to the code stay invisible to the rest of the ensemble until one chooses to publish them.

In order to address this, we propose a way of interacting on an even lower level: people should be able to write code together in parallel instead of only reshaping code sequentially.

A distributed collaborative editor allows patching both independently and in close collaboration, where every participant has full control over the entire patch.

In text-based environments this problem has been tackled in the last years[2], resulting in both commercial (like SubEthaEdit[6]) for the Mac OS-X platform) and FLOSS solutions (e.g. the cross-platform editor Gobby[1]).

However, all of these require dedicated server and client software, the latter usually being text editors (only), which makes it necessary to heavily tweak them, before being able to use them for collaborative patching with Pd.

The rest of the paper will discuss possible ways to allow multiple users edit a Pd patch collaboratively.

3. FIRST STEPS

In the performance series *Blind Date*[7], several people are creating a patch producing an audio-visual concert from scratch: A small group of performers (at least two, but usually no more than that) is working simultaneously on one patch, sharing their ideas but also interfering with each other, e.g. by hindering their partners to create a certain object.

The first attempt to fulfill this requirement was rather simple: several mice and keyboards were attached to the computer that runs the patch (figure 1).
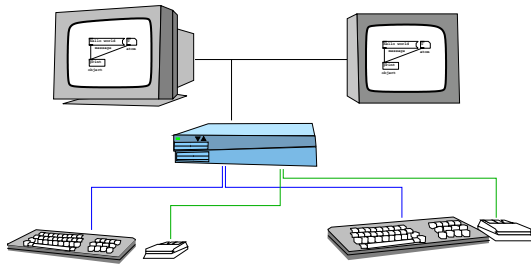

Figure 1: A simplistic collaborative patching environment, using USB-keyboards, USB-mice and a VGA-splitter

While this is solution is very simple to achieve, it has some obvious drawbacks:

- scalability: while connecting a number of input devices via USB is simple, distributing the video-output to several monitors quickly becomes cumbersome, as few machines have more than 2 VGA-outputs and VGA-splitters are expensive.

- locality: using USB- and VGA-connections the participants can only be so far from the host computer.

- focus: sharing keyboard and mouse has the strong disadvantage of having a shared focus. Therefore if one

person is working on one part of the patch, all other people are forced to work on the very same part too.

4. ARCHITECTURE

Pd has a networked architecture, that separates the DSP-engine (the `pd` process) from the user-interface (the `pd-gui` process). These two components communicate via a TCP/IP-connection.

When Pd is normally started, the DSP-engines starts up, opens a port (per default: 5600), and then starts a second process, the `pd-gui` which connects back to the engine (see figure 2).

Figure 2: Pd as it usually operates: the `pd-gui` process connects to the engine (pd dsp) on (e.g.) port 5600.

A naive approach to fix some of the above mentioned problems, is by introducing a proxy-server that sits in-between these two components. The only thing this proxy has to do, is to pass on all information it receives from the `pd-gui` to the engine and vice versa.
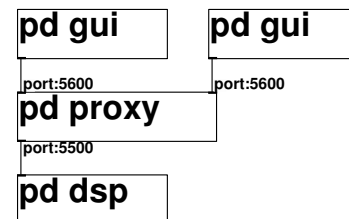

Figure 3: Several `pd-guis` connecting to a proxy which forwards the data to the engine. Data coming from the engine, is broadcast to all connected `pd-guis`

This can be extended to use several clients, e.g. `pd-guis`: all data from the `pd-guis` is sent "anonymously" to the engine (which does not know nor care which GUI it is receiving from). The engine does the interpretation of the data (e.g. create an object `[foo]` from the text "foo") and sends the display instructions (in theory "display an object called 'foo' with 1 inlet", but in reality rather "display the name 'foo' and draw 4 lines around it and then make a black area at position x/y") back to the proxy, which then distributes it to the `pd-guis`, updating all of them synchronously (see figure 3).

4.1 A simple proxy written in Pd

The good news about this approach is, that it does not need any special modifications to the Pd executables.

Both the Pd engine and the `pd-gui` can act as a server (opening a port on which they listen for incoming connections) and as a client (connecting to a port on a server), depending on their startup arguments.

Making the proxy a server for both engine- and GUI-clients allows (in theory) to connect multiple GUIs *and* engines together.

In a first proof of concept we just concentrate on connecting multiple GUIs to a single engine.

While the implementation of the proxy could be done in any environment, we chose to do a first realisation within Pd itself (see figure 4).

For the simplest case of a proxy that only relays the data between the engine and one or several GUI's, it is sufficient to use two servers (e.g. Martin Peach's `[tcpserver]` object[8]) and connect them to each other, sending all data that are received by one server to all clients connected to the other server and vice versa.

In order to allow GUIs to connect at a later time and still become synched with the rest of the patch, we use a buffer that stores all messages sent from the engine to the GUIs. Whenever a new GUI connects to the proxy, it is first updated with the entire history. Then it is synchronised with the rest and can start to act as a new replicated interface.

4.2 A more sophisticated proxy

A merely replicating proxy has several caveats. For instance, every user can modify the global state of the server, which can be problematic, when one of the users incidentally *Quits* "their" instance, which in fact quits the engine and furthermore all GUIs from the other users. This can be helped by adding filtering rules, that might modify or suppress certain messages (e.g. by not allowing the GUIs to send the `;pd quit` message to the engine, one can protect the engine against being shut down by one of the users).

It would also be possible to colourise parts of the patch based on the users who contributed to that part.

A proxy could also generate instructions on its own. For instance, it has proven problematic, that the participants are not able to see the mouse pointers of their partners.[1] A solution to this can be a proxy, that while passing on the mouse-movement to the engine also tells the connected GUIs to draw a mouse representation on their window(s), thus giving all participants visual feedback about all mouse pointers.

5. EXPERIENCE

Honestly, we haven't yet used the proposed solution in a "real world" environment. The *Blind Date* instance at the pd-convention07 will hopefully be our first public performance with this *networked* collaborative patching environment.

---

[1]The `pd-gui` gets the mouse position and forwards it to the engine, but it does not draw the mouse pointer. Instead, the mouse pointer is displayed by the X-server (or a corresponding service, depending on your platform). Since the X-server is local to the machine running the `pd-gui`, the performers have no visual feedback on what the other participants are currently doing with their mouse. This problem does not occur with keyboard interaction, since the characters are directly drawn by the `pd-gui`.

6. DISCUSSION

6.1 Similar environments for Pd

6.1.1 serendiPd

The proposed way of collaborative patching is very similar to that of Hans-Christoph Steiner's *serendiPd*[12] The main difference is, that *serendiPd* requires the clients to run a special patch for connectivity to the server, whereas here the clients do not even run a full-fledged Pd but only the GUI process. However, this also means, that the client does not produce any audio- or video- output on its own, which might be non-satisfactory in dislocated performances. In future versions the proxy might also support connecting multiple DSP-engines which might solve this, as each "node" would then run a synched version of both the engine and the GUI.

6.1.2 netpd

While Roman Häfeli's netpd[4] offers high level interaction, the here proposed solution is on a very low level. netpd concentrates on creating music with shared programs (by uploading pre-made abstractions and allow all users to control these abstractions in a collaborative manner), whereas we rather concentrate on creating programs/patches together. The ease and smoothness of netpd comes at the cost of obeying a strict set of rules, in order to make patches "netpd-aware". On the other hand, the proxy solution offers nothing but connecting.

6.2 Longing for a better separation

While Pd's separation between its engine and the user interface makes the proxy approach possible, one has to admit that things are not totally optimal. In the current implementation the GUI does hardly more than tell the engine what is happening at the input devices (keyboard, mouse) and in turn draws rectangles and characters as told by the engine. This leads to concurrency problems when several GUIs are communicating with one engine at the same time. For instance, if two users want to modify two sliders with their mice at the same time, the engine gets confused as it seems like one mouse pointer is jumping fast between the two sliders. If the GUI was able to evaluate the mouse movement by itself, it would be sufficient to tell the engine that the value of slider x has changed to y. This would eliminate the ambiguity of the two mouse movements, while at the same time reducing the total network traffic (and therefore the workload of the proxy).

7. CONCLUSIONS

In this paper we introduced a Pd-based environment for collaborative Live Coding, that allows interaction of several networked performers on a low level. Because no changes to neither Pd-engine nor `pd-gui` are required, the proposed proxy approach does not rely on any specific version of Pd or externals for both user interface and DSP. An implementation of this proxy done within Pd has been presented.[2]

---

[2]Note that unlike the engine and the GUI, the proxy imple-

8. ACKNOWLEDGEMENTS

9. REFERENCES

[1] A. Burgmeier and P. Kern. Gobby - a collaborative text editor, 2005-. Available from World Wide Web: `http://gobby.0x539.de/` [cited 2007/07/18].

[2] C. Cook. *Towards Computer-Supported Collaborative Software Engineering*. PhD thesis, University of Canterbury, Canterbury, New Zealand, 2006.

[3] A. de Campo, A. Vacca, H. Hölzl, E. Ho, J. Rohrhuber, and R. Wieser. Code as performance interface - a case study. In *Proc. of NIME*, New York, to be published.

[4] R. Häfeli. netpd, 2007. Available from World Wide Web: `http://www.netpd.org/` [cited 2007/07/20].

[5] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[6] M. Ott, M. Pittenauer, and D. Wagner. Subethaedit, 2005-. Available from World Wide Web: `http://www.codingmonkeys.de/subethaedit/collaborate.html` [cited 2007/07/18].

[7] pd graz. blind date. performance series, 2005.

[8] M. Peach. net library for pd, 2006. Available from World Wide Web: `http://pure-data.cvs.sourceforge.net/pure-data/externals/mrpeach/net/` [cited 2007/07/20].

[9] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, 1995.

[10] M. S. Puckette. Pure data. In *Proceedings of the International Computer Music Conference*, pages 224–227. International Computer Music Association, 1997.

[11] J. Rohrhuber and A. de Campo. Uncertainty and waiting in computer music networks. In *Proceedings of the International Computer Music Conference*, 2004.

[12] H.-C. Steiner. serendipd - impromptu networked collaboration, 2004. Available from World Wide Web: `http://at.or.at/serendipd/` [cited 2007/07/20].

---

mentation relies on a proper Pd version and certain externals installed.

DSP

t a <data to all DSP clients>

list prepend broadcast

list trim

tcpserver 5500

t a a <data from DSP clients>

GUI

t a <data to all GUI clients>

t a <data to single GUI client>

list prepend broadcast | list prepend send

list trim | list trim

tcpserver 5600

t a <data from GUI>

buffer

t f f

> 0

select 1

dump

fifop

list prepend 0

stores all messages from
the engine: whenever a
a new GUI connects, it
will be updated to the
current status.

new_connection

t b b

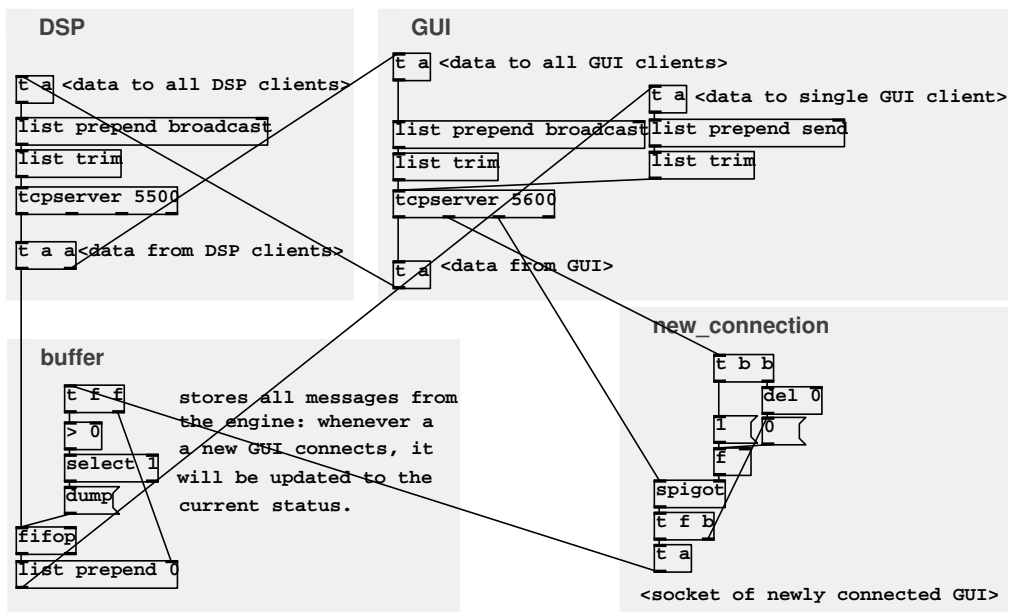del 0

1 | 0

f

spigot

t f b

t a

<socket of newly connected GUI>

Figure 4: A very simple replicating proxy