A JAVA-BASED REMOTE LIVE CODING SYSTEM FOR CONTROLLING MULTIPLE RASPBERRY PI UNITS

Oliver Bown
Design Lab
University of Sydney
oliver.bown@sydney.edu.au

Miriama Young
Music Department
University of New South Wales
miriama.young@gmail.com

Samuel Johnson
Design Lab
University of Sydney
sjoh7452@uni.sydney.edu.au

ABSTRACT

Cheap embedded devices create new opportunities for networked, distributed, generative or remote-controlled music. In this paper we present a simple audio programming environment designed to run realtime, remote live-coded audio on a low-cost completely wireless hardware setup consisting of a Raspberry PI, a WiFi dongle, a speaker and a battery pack. Audio is processed in realtime using the Beads library for realtime audio in Java, running on Oracle's distribution of Java for embedded devices. Code is remotely injected in realtime by sending Java class files over a socket connection to a dynamic class loader, which instantiates and runs the classes. We describe the system and its capabilities, and give an example of a performance that utilises this system. This paper is accompanied by a musical performance at ICMC 2013.

1. INTRODUCTION

The range and availability of cheap embedded devices has been accelerating in recent years, exciting audio developers with the growing plausibility of use-cases involving multiple low-cost audio computers. The release of the Raspberry PI in 2012 marked a significant development in this area. The Raspberry PI is a $35 computer capable of running a full Linux distribution, loaded from an SD card. Importantly it has USB ports and an analogue audio output. For a total cost of around $100 the PI can be expanded to include speaker, battery pack and WiFi dongle, turning it into a fully portable networked music generator (Figure 1). Alternatively, for roughly the same price it can be expanded with an audio dongle, switches and knobs to become a programmable effects pedal.

In this paper we consider the first configuration. In this configuration the PI has no audio input, but can be interacted with remotely over a network connection to control the realtime generation of sound. Although not necessary for most distributed music applications, the fact that the system is completely wireless extends its range of possible uses, given that it can be incorporated into miscellaneous portable objects. The ability to hide the device in objects without telltale hidden wires is also conducive to a sense of enchantment in interaction design. Finally the PI can be easily extended to further include other kinds of inputs and outputs, such as sensors and lights. This range of uses and interactive experiences is already covered by programmable smartphones but not at the same cost, programmability and configurability of the PI-based system.

The PI is easily configured so that on power-up, it automatically logs in, connects to a given WiFi network with a known network address, and loads an audio application. Thus, given a collection of PI modules, each module is ready to respond to dedicated control messages once activated.

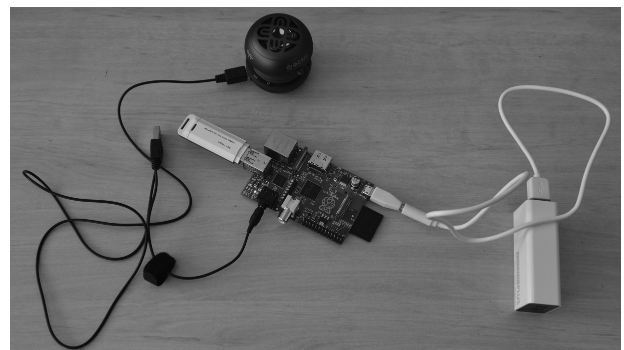

Figure 1. Portable wireless configuration consisting of Raspberry PI (centre), Moshi Bassburger self-powered rechargeable speaker (top), WiFi dongle (left of PI) and battery pack (right). Data is stored on an SD card (right of PI).

2. RELATED WORK

2.1. Composition for Distributed Electronics

Digital audio instruments form a field of experimentation with a lengthy history, that underlies current experiments with embedded devices for audio. The specific advance in recent years is the emergence of programmable consumer devices that enable programmers who are not specialised in embedded computing or electronics, such as the first author, to get straight to work. By hosting a full Linux distribution, the PI enables existing 'traditional' computer music work to be easily ported to the context of small, cheap, portable and low power devices that is typically the preserve of the electronics practitioner. The device is also general purpose, capable of running any executable software.

Many musicians have worked with modular electronic devices to produce distributed, networked generative and interactive musical works. In the world of hacked hardware, such as the work of Nic Collins [4], electronics are commonly distributed sonically, with localised sound sources, and may interact in simple ways or be remote controlled by a central device through wired or wireless connections.

Mobile wireless control interfaces have been used to creative effect through playful interaction, and the softening of the experience of computer music through apparently non-electronic interactive devices. The Australian improviser and experimenter Jon Rose has created a series of improvised musical works based around various types of ball[1]. In one case children play netball to an improvised electroacoustic soundtrack, data from the ball being part of the musical content. The work references Zorn's game-based improvisation strategies. In another case, a giant inflatable ball crowd-surfs the audience, providing a similar music control signal. In these cases the sound source is separate from the input device, and our present work effectively works in reverse, sending control signals to a portable sonic device.

The mobile phone is an obvious and ubiquitous sound-producing, portable, wireless device. Levin *et al.*'s *Dialtones: A Telesymphony*[2] was an early work exploring the creative potential of this distributed multiplicity of devices in the pre-smartphone era. More recently, the laptop orchestra and mobile phone orchestra have come to prominence, and a plethora of compositions, improvisations and interactive and generative works have been composed for this context [11]. A popular concept for these distributed electronic orchestras is the opportunity to physically distribute the sound as happens naturally in an acoustic orchestra [9]. Such performances often take the form of an instrumental ensemble, with one performer per device. Remote controlling large numbers of devices has been made possible by tools such as MultiVid[3] and can easily be managed using regular network connections, albeit with latency limitations in the case of wireless networking.

Audio-capable microelectronic devices such as the Beagleboard have been used in computer music experimentation such as by researchers at Stanford [2] and STEIM. A popular use of Beagleboards and Raspberry PIs by enthusiasts is in the area of building small custom instruments.

2.2. Live Coding

Collins *et al.* [3] present live coding as the realtime manipulation of program code in performance. Live coding provides an obvious solution to the demands of computer music improvisation, by utilising the broadest and most flexible interface available to a running audio application, namely the source-code itself. This enables any aspect of the running program to be modified, and for the dynamic creation of complex software elements such as audio signal graphs. A number of live coding tools and languages have emerged that cater for creative development and live performance, the most notable being SuperCollider[4], ChucK [10] and Impromptu[5]. One manifestation of live coding is as a virtuosic art-form. However it is also a practical way to get executable code onto a server and experiment with code in realtime without having to recompile and relaunch the program. Live coding is most suited to interpreted languages, but the realtime fast compilation provided by modern IDEs, coupled with dynamic class loading, means that compiled languages can also be live coded.

3. BASIC AUDIO ENVIRONMENT

At the time of the authors' initial experiments with audio on the PI, a number of realtime audio environments had been ported to it, with varying degrees of success. Any desktop environment already working under Linux is inherently capable of working on the PI, allowing for limits posed by RAM, CPU capability and a slightly reduced audio IO capability. The choice to use the Beads[6] audio library for Java, developed by the first author, is based on a preference for the Java language style and structure. Environments such as SuperCollider, which are real live coded environments built around a server-client model, are perfectly suited to this hardware context and are well developed and optimised over several years. For many users this will be a superior choice of environment. The purpose of the current research is not to propose a better solution or to make improvements on the live coding experience in environments such as SuperCollider, but rather to explore how networked remote control via real-time coding can be achieved in other commonly used creative contexts. The Java language and the derivative Processing creative coding environment[7], are extremely popular with creative

[1] See http://www.jonroseweb.com/
[2] See www.flong.com/telesymphony
[3] See http://marcotempest.com/screen/Public_MultiVid
[4] See http://supercollider.sourceforge.net/
[5] See http://impromptu.moso.com.au/
[6] See http://www.beadsproject.net
[7] See http://www.processing.org

digital art practitioners and the tools presented in this paper are directly applicable to these users.

Despite the added task of installing and running Java, setting up Java as an audio environment on the PI is straightforward and demonstrates reasonable performance. This is partly due to the fact that Oracle's recent distribution of Java for embedded devices is robust and efficient. In fact, establishing access to the audio hardware on the PI required no extra effort or set-up, and Java-based audio programs could be ported directly to the PI without issues. A common problem for realtime computation in Java is the risk of pauses caused by garbage collection operations. Newer Java garbage collectors have shown marked improvements, to the point where pauses are not likely to be a significant issue on modern computers. Nevertheless, it is helpful to be mindful that creating and destroying large number of software objects during execution may be a problem for audio software. For a wide range of audio applications, however, such as monophonic instruments, this is not a problem.

A basic audio processing test showed the PI capable of running stably with 10 oscillators at CD quality (44.1KHz, 16 bit, stereo signal output), with a signal vector size of 4096. This is a very modest capability compared to a typical laptop or desktop computer, but suitable for the musical use-cases imagined for this device. The sample rate was reduced to 22KHz and the signal to mono for added performance, without significant loss of quality (this is bearing in mind the fact that the speaker is itself lo-fi). The signal vector size was also increased to 8192 providing further headroom for processing delays. The downside of this is that audio processing commands are interpreted approximately every 200ms, far from the realtime control required by an instrument, but nevertheless suitable for more indirect control scenarios. The performance of embedded devices is increasing rapidly and these limitations will change accordingly.

4. NETWORK SETUP

The following configuration was used to maintain a robust and easily manageable connection between a master controller computer and a collection of PI clients. The master controller computer hosts a simple server, written in node.js[8] by the third author, that handles connections to any PI clients and sends commands to them in the form of executable class files (described below). PIs are set up to autoconnect to a predefined wireless router, *PINet*, as specified in their system configuration, and to autorun the `DynamoPI` application described below. Included in the application is a process that constantly notifies the server, located at a fixed IP address, of the PI's presence. A text file on the server computer can be used to maintain a list of known PIs, providing a user-defined name for the MAC addresses of any known devices. Devices with MAC addresses not stored in this list are autonamed. The server displays a list of all connected PIs, their connection strength (based on recent promptness of responses to pings), their names, MAC addresses and IP addresses (Figure 2). At present the server is not interactive, but it will be developed in future work to allow the user to send standard commands directly to PIs via a web interface. It is also proposed that PIs will send more detailed state information back to the server for display to the user, including a representation of their current audio graph. The live coder can choose which currently listed PIs to send commands to by specifying a list of recipient names in an array.

[8] See http://nodejs.org/

Dynamic Pi Master Server Control Panel

Name	ID	IP Address	MAC Address	Connection Status
pi6	0	10.0.1.4	00:9e:95:9c:4e:ec	good
pi9	1	10.0.1.8	00:9e:95:9c:47:ef	good
pi7	2	10.0.1.6	00:9e:95:9c:50:2d	good
pi8	3	10.0.1.7	00:9e:95:9c:4d:bc	good
pi3	4	10.0.1.12	00:9e:95:9c:50:93	good
pi4	5	10.0.1.5	00:9e:95:9c:47:e8	good
pi2	6	10.0.1.10	00:9e:95:9c:3f:b2	good
pi5	7	10.0.1.3	00:9e:95:9c:50:e2	good
pi1	8	10.0.1.11	00:9e:95:9c:51:0a	good

Figure 2. Browser-based interface to the PI Server.

5. DYNAMIC CLASS LOADING IN JAVA

The Java language includes a `ClassLoader` class that is capable of dynamically loading and linking previously unseen class definitions. This provides a way for new code to be introduced into a running Java program. A custom `ClassLoader` can be implemented that accepts the binary data representing a class file. The `ClassLoader` is able to define that class to the Java Virtual Machine (JVM). That class can then be used to instantiate an object using Java's *Reflect* API. Any given `ClassLoader` instance is not allowed to redefine a class with the same name and will reject new classes if they have the same name as existing classes. However, if the `ClassLoader` is destroyed and reinstantiated, the class definitions that it has loaded are removed. This allows classes to be dynamically reloaded, even when instances of those classes are still alive in the system.

Using sockets, it is then easy to send class binaries over a network to a running Java program, which interprets them as class definitions, defines them to the JVM and instantiates them.

6. A SIMPLE LIVE CODING ENVIRONMENT FOR BEADS

A simple Java application, `DynamoPI`, was written to run on the PI and allow remotely written Java code to effect realtime changes to a running audio graph. `DynamoPI` sets up a basic audio system consisting of an audio scheduler

(the class AudioContext from the Beads library) and a metronome (the class Clock from the Beads library), along with a general purpose storage area (the class Hashtable<String, Object> in Java), and, finally, the code required to set up a continuous socket listener to respond to incoming class definitions. In addition to DynamoPI, an interface, PIPO (standing for PI Play Object), was defined, containing a single method, action(final DynamoPI d); (Figure 3). PIPOs are the live-codeable component of the system.

```
public interface PIPO extends Serializable {
    void action(final DynamoPI d);
}
```

Figure 3. The PIPO interface.

For each incoming class received by the socket listener, any class implementing the PIPO interface is instantiated and has its action() method called.

For all incoming classes that are not of type PIPO, the class is simply defined by the ClassLoader but is not instantiated. This is because remotely written PIPOs may define anonymous or nested classes that also need to be sent to the PI for use by the PIPO class. For this to happen, at the sending end, all required classes need to be sent first, followed by the PIPO class. If a PIPO class is instantiated without its required classes being defined, a runtime exception is thrown. Once a PIPO class is received and run in this way, the ClassLoader is reinstantiated so that when code modifications to the PIPO are sent, the old PIPO class definition is not found and the PIPO gets redefined. As noted, this does not affect lingering PIPO objects, although it does mean that the definitions of classes required by those objects are lost.

Code can then be written into a new class implementing the PIPO interface on a master controller computer and sent through to the running DynamoPI program on the Raspberry PI, following the sequence of events shown in Figure 4. A development environment such as Eclipse can be used to provide a productive coding experience, with syntax highlighting, code completion, linking and linked documentation to all required libraries. The new PIPO can also be given a main() method that reads its own class file and sends it to the PI. Once this has been configured, running the PIPO on the controller computer causes the code in the action() method to be run on the PI. With Eclipse's plugin mechanism and code editor tools such as XText[9], it would be a relatively simple next step to make improvements to the Eclipse editor that would make live coding more convenient. For example, the editor could copy the design of the SuperCollider editor, in which a selected block of code can be run by pressing a key combination.

The step of reading and sending the class file involves some delay, and the execution of code is therefore not instantaneous. However, the setup does not preclude the

[9] See http://www.eclipse.org/Xtext/

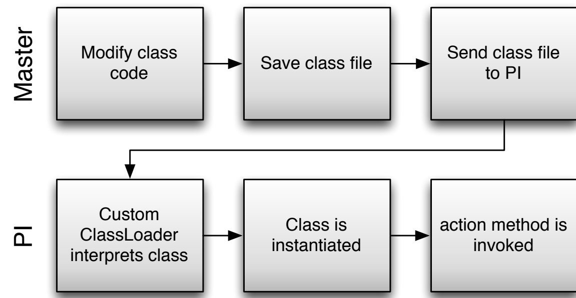

Figure 4. Steps between modifying executable code and it running on the PI.

possibility of other network connections that take care of timing, within the limitations of the WiFi network connection.

The Beads library contains a typical suite of signal generating and processing classes, utility classes for things like pitch management, a messaging system and methods for dynamically modifying the signal graph. For ease of use, the DynamoPI application class encapsulates a number of useful objects as well as wrapping them in a number of convenience methods. The class is passed to the PIPO's action() method so that PIPOs can act on the state of the audio system. Figure 5 shows the basic structure of the audio elements in DynamoPI. Sounds can be added upstream of a PolyLimit object, which can regulate how many audio elements are connected to it. Patterns can be added as event listeners to the main Clock object.

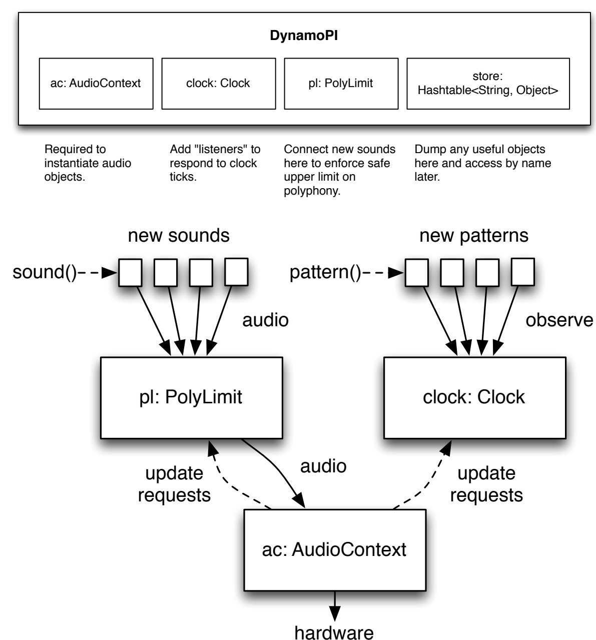

Figure 5. (Top) Classes provided by DynamoPI. (Bottom) Events leading from code to execution on PI.

6.1. An Example PIPO

An example of a simple `PIPO` program is given in Figure 6. This assumes that a group of sound samples has been preloaded into the `SampleManager`. The method `pattern()` attaches the specified anonymous callback class to the system's clock. At each tick of the clock, the `messageReceived()` method executes and a random sample is played. All audio objects in Beads have a method `kill()` that indicates to the audio scheduler that they should be removed from the signal graph. Once all references to an object in Java are removed the object is garbage collected. The default behaviour of the `SamplePlayer` class is to kill itself once it has finished playing the sample. Therefore the code example continues running without using up CPU or memory. All the `d.sound()` method does is to add the `SamplePlayer` to the signal graph.

```
public class DynamicPIPO implements PIPO {
    @Override
    public void action(final DynamoPI d) {
      d.pattern(new Bead() {
        public void messageReceived(Bead message) {
          Sample s = SampleManager.randomFromGroup("snd");
          SamplePlayer sp = new SamplePlayer(d.ac, s);
          d.sound(sp);
        }
      });
    }
}
```

Figure 6. A simple example of a `PIPO` program.

Both `pattern()` and `sound()` could also easily be modified to trigger `DynamoPI` to send back a state message to the master controller computer so that the user can monitor lists of objects connected to the clock and audio output. This offers the potential to write a GUI interface that would allow manual deletion or duplication of objects.

7. COMPOSITION AND PERFORMANCE

7.1. ChatterBox

Ten PI modules were built for use in live musical performance. The modules were not given permanent housings but can be easily incorporated into a number of objects, including everyday objects, bags, or balls. A space of approximately 7cm x 6cm x 5cm is needed if the various components are placed immediately next to each other.

In our initial performance the PIs were housed in origami boxes, (Figure 8) and distributed evenly throughout the audience. In the second context the boxes were suspended within an atrium, as an installation.

As a composer and musicologist, the second author's research focuses on the human voice and its interaction with technological media [12]. Her current interests lie in the realisation of a set of intimate, discrete listening pods for the creation of sound art that explores the voice and its mediation. The PI modules provide such a medium. In *ChatterBox*, an array of ten PI modules, housed in origami boxes, are each loaded with a spoken story.

The PI modules enable a multiplicity of discrete, simultaneous sounds to be distributed. They suggest particular compositional approaches to sound materials and their dissemination. For one, the PI modules enable a paradoxically intimate listening environment within the public space. The small size of the module speakers limits the available frequency range. Further, because of the module's tactility and relatively small size, the listener is encouraged to engage in an enactive listening experience: to pick up and press a module to their ear in order to fully apprehend the sound within. Due to this set of constraints and possibilities, the set of PI modules housed within boxes lend themselves well to the spoken voice, and to storytelling in particular. For *ChatterBox* these were sourced from public domain readings of stories by Charles Dickens, thanks to LibriVox[10] and its narrators.

In this context, the PI modules naturally suggest the act of eavesdropping, that popular human pastime of lending one's ear to become privy to secrets otherwise out of range, as discussed by Paul Lansky in [13]. While the listener is invited to audition each box individually, the effect of hearing all units simultaneously creates a 'wash' of babbling sound, similar to the multiplicity of voices heard at a cocktail party (see Paul Lansky and Francis Dhomont in [1]). *ChatterBox* exploits the listener's ability to 'zoom' in and out of different modes of listening, and to oscillate between the singular voice and the ten as a polyphonic whole.

As befits the Twitter era, the spoken stories in *ChatterBox* are a set of overlapping one-way monologues. An individual story is often lost in the digital deluge, projected to an imagined but impossible audience of everyone and no-one simultaneously. Reflecting the noise of social media, *ChatterBox* attunes our ears back to the special intimacy of the storytelling experience.

Each told story is mediated through a temporary voicebox. The notion of a speaking box or voicebox connects to the idea of the larynx as a speaking instrument, and to the long technological history of early speaking machines that attempted to emulate the voice [5, 8, 6]. Although the boxes are neutral in appearance, the signifying voice enables the box to take on particular anthropomorphic qualities. The listener may imbue these small speaking boxes with a personality, based on the sound of the storyteller's voice - woman/man, young/old, calm/agitated, large/small, happy/sad and so on. In performance, the boxes could be apprehended as a set of reconstituted Dickensian characters.

The piece also calls attention to the inherent musicality of the voice. This is brought into focus through subtle electronic manipulation of the spoken voice so that it oscillates between the discernible and the abstract. The storytelling voices are 'tuned' to specific tones through ring-modulation. This effect is scaled gently so that the voice from each box gradually becomes a sine tone, and vari-

[10] See http://librivox.org/

```
////// PART 3 /////
//back to babble

time += 1f * 60 * 1000;

timedEvent(d, time, new Bead() {
    public void messageReceived(Bead message) {
        rfreq.clear();
        rfreq.addSegment(100, 50);
        rwet.clear();
        rwet.addSegment(0, 2000);
        masterGain.clear();
        masterGain.addSegment(0, 2000);
        masterGain.addSegment(0, 2000f, new Bead() {
            public void messageReceived(Bead message) {
                d.put("s", "babble");
                //clean up anything else
                pos = new Long(0);
                d.put("p", pos);
                looperStart.setValue(-1);
            }
        });
        masterGain.addSegment(sampleGain, 100f);
    }
});
```

```
////// PART 4 /////
//loops - stop first, then go to loop (for group 0 and 1 only)

time += 3 * 60 * 1000;

float timeRandomnessRange = 30000;

if(group[id % group.length] != 2) {
    timedEvent(d, time + d.rng.nextFloat() *
            timeRandomnessRange, new Bead() {
        public void messageReceived(Bead message) {
            masterGain.clear();
            masterGain.addSegment(0, 100f);
            masterGain.addSegment(0, 3000f, new Bead() {
                public void messageReceived(Bead b) {
                    gainLoop(gp);
                    looperStart.setValue(20000);
                    looperFreq.setValue(
                            1000f / (ratios[id % ratios.length] * 2000f));
                }
            });
            masterGain.addSegment(sampleGain, 4000f);
        }
    });
}
```

Figure 7. A small part of the *ChatterBox* composition.

ous sine tones are stacked to create 'harmonies'. Looping is also applied to the voice, creating different rhythmic effects between boxes. Throughout the piece a series of such interventive transformations is applied to the babble, exploring the potential of the multiplicity of voices to come together in brief episodes of unity, before returning to a discrete crowd scene. A small segment of the *ChatterBox* composition code is shown in Figure 7.

7.2. Deployment

The work has been presented so far in two contexts. In the first case the PI modules were placed in their boxes amongst the seating at a concert venue, and the piece was "performed" in the liminal space before the start of the concert proper. The piece began as audience members took their seats, and the initial babble of voices slowly faded into their conversation. Some listeners moved about, to explore the behaviour of sound around the space. Curious listeners picked up the individual boxes to audition specific stories. Once the listener could focus on the individual narrator, they became drawn into the individual storytelling experience. As their listening zoomed away from the individual box, the sonic experience returned to a flurry of babble. The subsequent subtle manipulation of sounds created a situation in which most of the audience could continue their conversations but be occasionally drawn into the composition by specific events, such as the emergence of distributed harmonic tones. In this performance context no attempt was made to hide the electronics and the listeners' enchantment was coupled with natural technical curiosity.

ChatterBox was presented as an unannounced intervention located in a concert setting but taking the form of an installation. Rather than being placed remotely on the concert stage, the sounding modules were situated amongst the audience where they could be handled. Due to the presence of the speaking voice embedded both within the modules and amongst a chattering audience, the installation remained somewhat hidden, and subsequently announced its presence more strongly whenever the audio was manipulated into tones. This played on an ambiguity about whether the piece should be apprehended as a formal performance or as the background sound of an ongoing installation. The arrival of coherent tones brought a sudden silencing to the audience, and the subsequent return to the spoken voice curiously permitted the audience to engage in their own conversations again. Thus, the project blurred the distinction between performer and audience/listener; between concert and installation; between one's own voice box and the chattering, un-embodied boxes that surrounded the audience.

In the second context, the PI modules were suspended in the atrium of Sydney Conservatorium of Music, positioned over a large staircase and run as an ongoing interactive installation. A Kinect depth sensor was used to track movement within the space. The same *ChatterBox* composition was altered so that interaction events picked up by the Kinect triggered the master computer to send different blocks of code to the PIs. In this case the software was not live-coded in a performative sense. This dynamic control made the development and deployment of the piece simple and flexible.

8. SUMMARY AND FUTURE WORK

This paper demonstrates a user-friendly approach to controlling realtime audio on a low-cost audio platform, including the ability to inject new code into an already running program using the popular Java language. The PI modules will be refined through optimisation, and improvements to the software and interface. Immediate steps include optimisation of the audio system, increasing the robustness and functionality of the audio server, providing more detailed feedback on each PIs state in the server interface, such as the current structure of its audio graph, adding interactive control to the server interface, and providing additional tools such as a software update script and other scripts for loading sound samples onto the PIs. The addition of an audio graph representation and inter-

active control to the server interface could be combined so that audio elements such as oscillators or sample players automatically prompt the generation of controller widgets on the server. The PIs are also easily extended to include sensors and actuators, supporting additional use cases. However for the time being remote control from a central server is our main area of interest. As well as the high audio latency, control over WiFi introduces additional unavoidable latency.

The system provides a context for further experimentation in creative performance, installations and sound design:

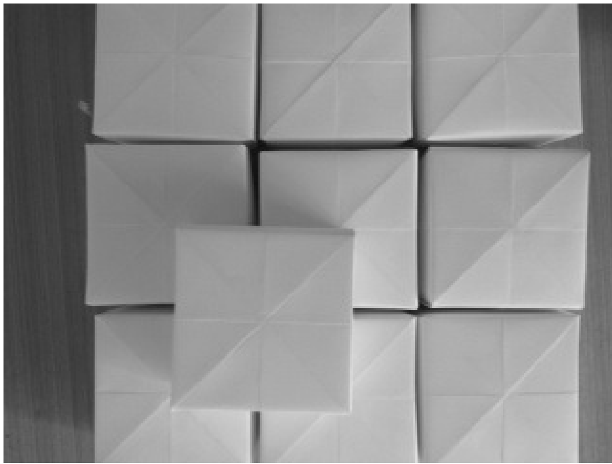

Figure 8. Origami boxes housing PIs.

- **Simple Performance Add-on**: An existing live configuration can easily be extended to include the PIs since they do not require the routing of audio, only the sending of messages that trigger remote responses on the PIs. Commands to play audio can easily be sent to client PIs in synch with on-stage music elements.
- **General Purpose Portable Sound Tools**: The system makes for an extremely versatile tool for theatre performances and art installations, being easily placed in physical spaces, on stage, amongst an audience, or within installation elements.
- **Ubiquitous Sound Design**: The kit can easily be used to experiment with sonic enhancements to everyday objects such as kitchenware, doors and pot plants, under the research agenda of ubiquitous sound design.
- **Multiple Devices, Multiple Controllers**: The PIs join a growing ecosystem of existing networked, sound-making computational devices including computers, phones and tablets. The proposed networked coding environment is one of a number of possible solutions that would allow multiple sound devices and multiple controller devices to adaptively connect via a central server. Further developments will be made to the interface that will allow any networked device to dynamically join a performance system as either output (screen or speaker), controller, or both.

With respect to the last point, a further challenge lies in finding suitable programming methods for easily dealing with variable numbers of devices. A performance might be designed for n devices, with $5 \leq n \leq 100$, but such that the behaviour scales appropriately to the number of modules. Hespanhol *et al.* [7] describe this for interactive artworks using the term *elasticity*. We expect to see increased efforts in developing such creative programming environments in the near future.

9. REFERENCES

[1] U. Aumüller, "My cinema for the ears," DVD, Bridge, 2002.

[2] E. Berdahl and C. Chage, "Autonomous new media artefacts (AutoNMA)," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Oslo, Norway, May-June 2011.

[3] N. Collins, A. McLean, J. Rohrhuber, and A. Ward, "Live coding in laptop performance," *Organised Sound*, vol. 8, no. 3, pp. 321–330, 2003.

[4] N. Collins, "Searching for the perfect beep – a personal history of hardware hacking," Accessed online 21/2/2013 http://www.nicolascollins.com/texts/perfectbeep.pdf, 2006.

[5] P. R. Cook and C. N. Lieder, *SqueezeVox: A New Controller for Vocal Synthesis Models*. Ann Arbor, MI: MPublishing, University of Michigan Library, 2000.

[6] H. Dudley and T. H. Tarnoczy, "The speaking machine of Wolfgang von Kempelen," *Journal of the Acoustical Society of America*, vol. 22, pp. 151–166, 1950.

[7] L. Hespanhol, M. C. Sogono, G. Wu, R. Saunders, and M. Tomitsch, "Elastic experiences: designing adaptive interaction for individuals and crowds in the public space," in *Proceedings of the 23rd Australian Computer-Human Interaction Conference*, ser. OzCHI '11. New York, NY, USA: ACM, 2011, pp. 148–151. [Online]. Available: http://doi.acm.org/10.1145/2071536.2071559

[8] J. Sterne, *The Audible Past: Cultural Origins of Sound Reproduction*. Duke University Press, 2003.

[9] D. Trueman, P. Cook, S. Smallwood, and G. Wang, "Plork: The princeton laptop orchestra, year 1," in *Proceedings of the 2006 International Computer Music Conference*, 2006.

[10] G. Wang, "The ChucK Audio Programming Language: An Strongly-timed and On-the-fly Environ/mentality," Ph.D. dissertation, Princeton University, 2008.

[11] G. Wang, G. Essl, and H. Penttinen, "Do mobile phones dream of electric orchestras?" in *Proceedings of the 2008 International Computer Music Conference*, 2008.

[12] M. Young, "Singing the body electric," Ph.D. dissertation, Music Department, Princeton University, 2007.

[13] ——, *Singing the Body Electric - The Human Voice and Sound Technologies*. Ashgate, forthcoming.