



SEAMUS 2011

FROST SCHOOL OF MUSIC
UNIVERSITY OF MIAMI

Conference Paper

Presented at the 26th Annual Conference of the Society for Electro-Acoustic Music in the United States
January 20–22, 2011
University of Miami Frost School of Music
Miami, Florida

Papers presented at SEAMUS 2011 have been blindly peer reviewed by members of the paper selection committee on the basis of a submitted abstract. The paper presented here is reproduced directly from the author's or authors' manuscript without editing or revision by the conference committee.

Distributed Performance Systems using HTML5 and Rails

Dr. Jesse Allison¹

¹ Louisiana State University, Baton Rouge, LA, 70806, USA
jtallison@lsu.edu | jesse@electrotap.com

ABSTRACT

Distributed performance systems present many challenges to the artist in managing performance information, distribution and coordination of interface to many users, and cross platform support to provide a reasonable level of interaction to the widest possible user base.

Now that many features of Rails 3 and HTML 5 are becoming solidified, powerful browser based interfaces can be utilized for distribution across a variety of static and mobile devices. Development has focused on canvas based javascript control interfaces, leveraging the power of a supporting Rails web application to handle distributed user functionality and pass interactions via OSC to and from realtime audio/video processing software. Interfaces developed in this fashion can reach potential performers by distributing a unique user interface to any device with a browser anywhere in the world.

1. INTRODUCTION

Distributed performance systems have undergone many leaps forward with the increased speed of networks and the emergence of ensembles such as laptop orchestras that are ideal for exploring this kind of interaction. As mobile devices become more powerful and ubiquitous, integrating them into performance systems is increasingly desired. Unfortunately, there are many varieties of operating systems, development kits, and proprietary code, making the incorporation of such devices difficult to do except under very narrow use cases - essentially requiring a platform dependent application to be built for each device in the system. Also, managing direct communications between more than just a few devices becomes incredibly complex.

Now that many features of HTML 5 are being incorporated among the leading browsers, powerful browser based interfaces can be utilized for distribution across a variety of static and mobile devices - making worldwide collaborative creative arts a distinct possibility. Development has focused on a variety of interactions, direct and indirect, including canvas based javascript control interfaces. These control devices leverage the power of a supporting Ruby on Rails web application to handle the distributed user functionality and pass communications via OSC to and from MaxMSP, a realtime audio/video processing language. Interfaces developed in this fashion can reach potential performers by distributing a unique user interface to any device with a browser anywhere in the world. In overcoming platform dependency issues, the emerging viability of browser based interface is an exciting frontier.

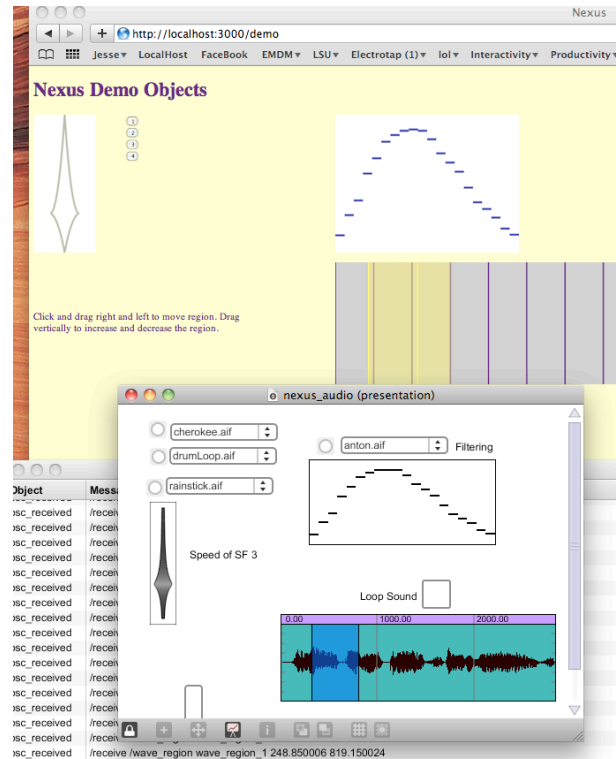


Figure 1 Browser interface objects and controlled Max application.

2. DISTRIBUTED PERFORMANCE SYSTEMS

For our purposes we will define distributed performance systems as an instrument whose control interface has been divided and distributed among potential performers. The audio rendering engine may be centralized or distributed along with the interface. This type of system enables collaborative performance and would encompass most network performance systems and collaborative music systems.

Although this kind of system could and should be used in many types of net art contexts, I am particularly interested in the live performance aspect of such a system. As such the responsiveness of the interface is a primary concern

2.1. Challenges to Overcome

The creation of such a system has two main challenges: platform dependency, which limits the number and type of potential participants, and distribution management, which becomes unwieldy very quickly. Other subsidiary challenges include responsiveness of the interface, security of the networked system, delay in streaming audio if utilized, and normal network configuration issues.

As a creator of a distributed system, one must limit oneself to coding a client application on a small number of platforms, usually one. As the artist adds platforms to incorporate larger numbers of users, they have to deal with platform dependent code and frameworks, various programming languages, separate development environments, not to mention the difference in graphical code. A simple application designed to reach both the iOS and Android devices would have to be programmed in Objective-C and Java, use XCode/Interface Builder and Eclipse, device dependent methods for event handlers, data management, network communication, and device peripheral integration, and 2 separate graphical drawing frameworks. This is incredibly unwieldy and pushing this same application to a blackberry, PalmOS, or windows mobile device would create a development and support nightmare.

Distribution management is another complex issue. As the instrument's interface is divided and passed out among participants, a method for coordinating each control node's parameter changes and commands must be coded. If the instrument is centralized, then commands being sent back to the system must be mapped from individual control nodes to their appropriate places. If it is to accommodate a fluctuating number of users, it has to coordinate the dynamic assignment of parameters to incoming users,

creation of an individualized graphical user interface, and parsing of the returning interactions. In the end the design of the UI distribution system has to be dynamic for the number of users, the parameters, and the GUI, which is very difficult to code successfully.

One increasingly viable solution to our two primary problems is the use of a web application and browser based interfaces to distribute and manage user interaction.

2.2. Browser Based Interaction and HTML5

Due in large part to the ubiquitous usage of the internet, much effort has been poured into creating standards for display and interaction with web pages. Although browsers are notorious for behaving somewhat differently, overall they can be made to look and function similarly across a wide variety of platforms both desktop and mobile. Compared to other cross platform initiatives, browser functionality is quite well developed.

Javascript UI initiatives such as prototype, jQuery, and Scriptaculous are great starting points for user interactivity, but have focused primarily on solving the problems of browser interaction with the server and css augmentations. These can be used as the backend for graphical musical objects, but are not satisfactory for many musical interfaces. HTML5 proposes specification of a number of tags that will be quite useful for the development of viable performance user interfaces. The <canvas> tag provides a 2 dimensional drawing space that can be tied to javascript and AJAX events to make viable user interface objects. A 3D specification for <canvas> is in active development as well, further expanding the interface possibilities.

To use the canvas with javascript is a fairly simple procedure. One must declare the canvas in the

DOM and specify through css any associated settings (Figure 2). Then in a javascript file or script tag, associate the canvas object and define the drawing context (Figure 3), attach event handlers (Figure 4), define methods to handle the events and draw the user interface, and create an AJAX response to send interactions to the server (Figure 5).

```
<canvas id="slider_1" width="100" height="225" >[Slider]</canvas>
```

Figure 2 Declare Canvas

```
var slider_sketch = document.getElementById("slider_1");

slider_draw();

function slider_draw()
{
    var slider_sketch_context = slider_sketch.getContext("2d");
```

Figure 3 Associate Canvas Object and Define Drawing Context

```
slider_sketch.addEventListener("mousedown", sliderOnMouseDown, false);
slider_sketch.addEventListener("mousemove", sliderOnMouseMove, false);
slider_sketch.addEventListener("mouseup", sliderOnMouseUp, false);
slider_sketch.ontouchstart = sliderOnTouchStart;
slider_sketch.ontouchmove = sliderOnTouchMove;
slider_sketch.ontouchend = sliderOnMouseUp;
```

Figure 4 Attaching Event Handlers

```
function slider_passValue( e) {
    new Ajax.Updater('slider_info', '/slider_move',
    {asynchronous:true, evalScripts:true,
    parameters:'slider_number=7&slider_value=' +
    encodeURIComponent(1.- (slider_val/slider_height)) })
}
```

Figure 5 AJAX Response

This process is similar to GUI development on most platforms today.

2.3. Ruby on Rails

Ruby on Rails is a framework for developing web applications that handles much of the distribution and information management issues through well-trodden conventions as opposed to configuration. The end result is a very rapid development

process that is easily adaptable to the developers specific goals, and it handles many of the normal issues of scalability, testing, database configuration and management, and customized user interface generation.

It is structured around the prevalent model, view, controller (MVC) architecture. The model manages all of the data for the application. This can be direct information like parameters and instrument states, or meta-data for files stored elsewhere. Ideally, the model handles all access to this information providing methods for creating, reading, updating, and deleting as well as accessing aggregates of the stored information.

Some implementations of a performance system in rails may make good use of the Model for handling artistic information, but most of the action is held in the View and Controller stages.

2.3.1. Controller

The controller handles all incoming requests. In our usage case, this would either be a prospective user requesting a portion of the collaborative interface or the passing of UI changes back to the server. A separate method is created for each case of interaction. The controller can then identify who is sending the information based on session data or parameter names, and respond accordingly. In the instance of someone joining the performance system, it would decide what interface objects to disseminate and pass this information in variables to the view to be rendered. Note that because of the sensible conventions utilized in the Rails Framework, unless you need unique logic to handle a request, much of this code is handled for you as is evidenced in the code to create the sliders page (Figure 6).

```
def sliders
```

end

As interface movements come back to the controller, the controller can parse the incoming data as needed and then send the appropriate information to the audio rendering engine through OSC. In the example below, the ROSC library is utilized (Figure 6).

```
def sliders
end

def slider_move
  slider_number=params[:slider_number]
  slider_value=params[:slider_value]
  m = OSC::Message.new('/slider', nil, slider_number.to_i, slider_value.to_f)
  OSCSender.send(m, 0, OSCHost, OSCSendPort)

  render :nothing => true
end
```

Figure 6 Typical Controller Logic Handling a Slider View and Response

2.3.2. View

The view handles the presentation of the User interface. The controller passes the view any variable information on a per request basis. The view uses embedded ruby to create text and markup tags and send variable data to each incoming request so that a button sent to one user may send back a response as button_1 while another user would have button_2, etc.

```
<div id="buttons">
  <%= button_to "1", button_press_path('1'), :remote => true, :method => 'put' %>
  <%= button_to "2", button_press_path('2'), :remote => true, :method => 'put' %>
  <%= button_to "3", button_press_path('3'), :remote => true, :method => 'put' %>
  <%= button_to "4", button_press_path('4'), :remote => true, :method => 'put' %>
</div>

<canvas id="multislider_1" width="300" height="225">[Multi-slider]</canvas>

<canvas id="slider_1" width="100" height="225" >[Slider]</canvas>
```

Figure 7 Embedded Ruby (ERB) to Customize the HTML Code

The view also handles distributing the associated javascript interface code described above. This can again be customized on a per user basis using embedded ruby.

2.4. Audio Engine (or other performance generating engine)

The final link in the chain is an audio rendering engine of some sort. I used MaxMSP receiving OSC messages through [udpreceive]. Alternatively one could use Ableton Live, PD, Supercollider, Chuck, or any OSC capable audio engine.

One interesting case concerns the development of the Jamoma Audio Graph. In a recent incarnation, cross compilation was added allowing one to create an audio graph in Max and export it as a pd patch, c++ code, or ruby. In experiments with Timothy Place, we were able to get a Ruby Jamoma Audio Graph running inside a rails application which would allow for a web service to be deployed which would handle all of the audio production as well as user interaction.

3. CONCLUSIONS AND FUTURE DIRECTIONS

Browser based user interactions are a viable option for distributing performance interfaces to many users. In small numbers of performers and local network situations, the responsiveness can be used for immediate real-time interactions. As the number of users and the network scope expands, techniques will have to be employed to retain expected responsivity. However the many other problems of scalability are remediated using the Rails web service approach.

Now that viability of this method has been established, many avenues of exploration present themselves. A library of javascript user interface objects will be developed to facilitate the ease of use and incorporation into Rails applications and other web technologies.

The possibility of enabling MaxMSPs jsui object to make use of the core drawing and interactive code of the javascript library will also be explored.

On the web application front, example rails applications should be developed for varying use cases including small ensemble with direct responses, broad distribution utilizing audio streaming, location awareness, and control parameter database, and picture/drawing based interactions. Depending on interest, once examples are in place, an open source initiative will be released.

Viewed December 2010 at <http://createdigitalmusic.com/2010/10/libpd-put-pure-data-in-your-app-on-an-iphone-or-android-and-everywhere-free/>

4. ACKNOWLEDGEMENTS

This work was supported by the Louisiana State University Center for Computation and Technology, LSU AVATAR Initiative, LSU College of Music and Dramatic Arts, and Dr. Stephen David Beck.

5. REFERENCES

- [1] Barbosa, A. *Displaced Soundscapes: A survey of Network Systems for Music and Sonic Art Creation* Leonardo Music Journal, vol.13, 2003, 53-59.
- [2] Mills, R. *Dislocated Sound: A Survey of Improvisation in Networked Audio Platforms* In Proceedings of New Interfaces for Musical Expression Conference, (Sydney, Australia, 2010)
- [3] Place, T., Lossius, T., Peters, N. *THE JAMOMA AUDIO GRAPH LAYER*. In Proceedings of the 13th Int. Conference on Digital Audio Effects (DAFx10), (Graz, Austria, Sept. 6-10, 2010)
- [4] Kirn, P. *libpd: Put Pure Data in Your App, On an iPhone or Android, and Everywhere, Free.*