Audio Engineering Society

Convention e-Brief

Presented at the 133rd Convention
2012 October 26–29 San Francisco, CA, USA

EspGrid: A Protocol for Participatory Electronic Ensemble Performance

David Ogborn[1]

[1] McMaster University, Hamilton, Ontario, L8S 4M2, Canada[2]

ABSTRACT

EspGrid is a protocol developed to streamline the sharing of timing, code, audio and video in participatory electronic ensembles, such as laptop orchestras. An application implementing the protocol runs on every machine in the ensemble, and a series of "thin" helper objects connect the shared data to the diverse languages that live electronic musicians use during performance (Max, ChucK, SuperCollider, PD, etc.). The protocol/application has been developed and tested in the busy rehearsal and performance environment of McMaster University's Cybernetic Orchestra, during the project "Scalable, Collective Traditions of Electronic Sound Performance" supported by Canada's Social Sciences and Humanities Research Council (SSHRC), and the Arts Research Board of McMaster University.

1. CONTEXT

EspGrid is both a protocol, and a software application implementing that protocol, aiming to streamline the sharing of timing, code, audio and video in participatory electronic ensembles, such as laptop orchestras[1]. The software has been initially developed in close connection with the regular rehearsal and performance schedule of a specific laptop orchestra, the Cybernetic Orchestra at McMaster University in Hamilton, Canada[2]. In the Cybernetic Orchestra, each member performs with their own laptop and loudspeaker, with sole responsibility for their own sound performance, although each member is connected to all others via a local area network (LAN). The performance practice of the orchestra has two main features: live coding[3], in which each member performs by writing/modifying programs in the on-the-fly-programming language ChucK[4], and the use of synchronized beat structures.

2. ARCHITECTURE

2.1. No servers

In the Cybernetic Orchestra, the control of centralized musical processes rotates freely and democratically. The EspGrid architecture reflects this social practice through the absence of any central server. Every machine in the ensemble (every "node") runs the EspGrid application, implementing the same behaviour.

2.2. Multiple machines to each human

A given member of the orchestra may be performing with multiple nodes - for example, a laptop that is being used to run demanding software synthesis patches, as well as a tablet that is being used to control the beat system and chat with fellow orchestra members. To the other members, both nodes will appear as extensions of the same human performer. In the application's Preferences, participants identify themselves with a <name> (i.e. david) and then identify the particular <machine> (i.e. laptop).

2.3. The private protocol

EspGrid actually sets forth two protocols, a private protocol and public protocol. Each node in the ensemble runs the EspGrid application, and each instance of the application communicates with all others over the network according to the *private protocol*. Electronic performers do NOT need to know or understand anything about the private protocol – it works "automatically" and behind the scenes to provide the sharing and co-performance resources of the grid.

2.4. The public protocol

On each node the EspGrid application communicates with sound and music performance applications (such as Max, ChucK, SuperCollider, PD, etc.) via Open Sound Control (OSC)[5] messages sent to local UDP network sockets. This is the *public protocol*. Performers never address other nodes on the network directly. Instead, they interact with their local EspGrid application (directly or via the public protocol), while the application interacts with the other nodes.

3. LOW-LEVEL RESOURCES

The interaction of EspGrid applications forms two low-level resources upon which higher-level resources (closer to the human intentions and structures of participatory electronic performance) depend: a list of the other nodes on the grid, and a synchronized clock.

3.1. The list of nodes

When a new instance of the application is launched, it broadcasts a beacon message many times throughout the first few seconds, in an effort to make sure that new additions to the ensemble are noted as quickly as possible. Thereafter, each instance transmits the beacon once every several seconds. The receipt of a beacon is used to populate and maintain a list of nodes, including the identity of each <name> and <machine> plus other information. The receipt of a beacon also causes a node to respond by broadcasting an acknowledgement. When a node receives an acknowledgement of its beacon it uses this to estimate the latency introduced by the network, and then keeps track of the lowest observed latency between itself and each other node (i.e. the fastest possible return trip between each pair of nodes).

3.2. The synchronized clock

Forming a synchronized clock signal across the grid allows higher-level resources to determine the sequence of multi-user actions, and to synchronize or sequence effects (for example, beats) across the grid. Every beacon contains the current value of the issuing node's synchronized clock. If a node receives a clock value that is higher than it's own current clock, it increments an adjustment factor so that it's own clock "races ahead" to tie with the received clock value. A second adjustment helps to compensate for network latency. Half of the minimum latency with a specific node (as recorded in the list of nodes) is used to adjust the received clock values (cf. Cristian's algorithm[6]). For example, if the latency to node B as measured by node A is 10 ms, whenever node A receives a BEACON and clock value from node B it will add 5 ms to that clock value before any comparison.

4. HIGH-LEVEL RESOURCES

On the basis of the low-level resources formed by the EspGrid protocol, a number of high-level shared resources can be formed, relating directly to the practices of performers in a participatory electronic ensemble. Currently stable high-level resources include the sharing of synchronized beats, the sharing of code fragments during live coding, and a basic chat protocol for text-based communication.

4.1. Beat sharing

4.1.1. Problems with "naïve" beat synchronization

Compositions or improvisations using synchronized beats are common with laptop orchestras, but the code for these pieces often employs an all too simple method for synchronizing beats, wherein one machine simply issues triggers via UDP on the local network. This has three serious problems, all of which are strongly in evidence with the commercial WiFi hardware commonly used by laptop orchestras: (1) packets may be lost completely, (2) packets will arrive with a minimum latency, leading to a discrepancy in timing between the issuing machine and receiving machines, and (3) the latency varies greatly from packet to packet (jitter), leading to late beats.

4.1.2. The temporary solution

Various "short-term" solutions to the problem of UDP jitter and lost packets exist and were employed in the orchestra during the earliest phases of this research:

- Many identical beat messages can be issued in rapid bursts, in order to decrease the impact of lost packets.
- Some of the timing can be delegated to the receiving machines, in order to make jitter less noticeable. For example, a whole bar or section of beats can be triggered by a single network message.

Neither of these methods, whether alone or in combination, leads to completely solid synchronized beats. Neither compensates for the minimum latency between the sender and receiver.

4.1.3. Shared beat parameters

EspGrid implements shared beats by sharing the parameters of the beat structure across the grid. The content of the parameters, rather than the timing of their arrival, determines the timing of beat events. An Objective-C class called EspKeyValueController allows key-value pairs to be synchronized across all nodes. When user action changes a parameter, that action and the resulting value are time-stamped (according to the shared, synchronized clock established as a lower level resource) and later user actions trump earlier ones. Every machine running the EspGrid application has GUI elements that allow the beat parameters (tempo, how many beats in the bar, active/inactive) to be displayed and modified. If one user changes a parameter on their machine, all other users see and experience the change. Not only is the beat shared – so is the method of controlling that beat.

4.2. Code sharing

In a participatory laptop orchestra, open to the widest possible pool of potential members, informal learning is both a prized and frequent occurrence. EspGrid provides a technical accelerant to informal learning through a facility for rapidly sharing code fragments. The application shows a constantly updated list of all the code fragments that have been shared by other nodes, from which any piece of shared code can be grabbed and then pasted into a live coding window.

The default mechanism for sharing code is to copy it to the clipboard then click a button in the EspGrid application. However, in the Cybernetic Orchestra we also use a modified version of the miniAudicle[7] in which every successful execution of a new code "shred" leads automatically to the sharing of that code.

4.2.1. ANNOUNCE – REQUEST – DELIVER

In the private protocol, the code sharing mechanism involves the following basic messages: (1) When a node is going to share a code fragment, it *announces* the existence of the code fragment to all of the other nodes; (2) When a node wishes to acquire a code fragment, it *requests* the code fragment; and (3) When a node sees that one of it's fragments has been requested, it *delivers* the content of the fragment in broadcast messages. All nodes receive the content and store it, thus avoiding some duplicate requests.

4.2.2. Further consequences

The Cybernetic Orchestra is just beginning to explore the possibilities this affords in terms of new types of improvisation. For example, code sharing suggests improvisation structures in which players are required to successively modify each other's code during live coding. Additionally, and especially when automatic code sharing mechanisms are engaged, EspGrid provides a novel form of documentation of collective live coding performances: the database of all code successfully executed during a given performance.

4.3. Chat

Although technically quite simple, the chat feature of the EspGrid protocol is crucial to many performances by the Cybernetic Orchestra. Every person on the grid can send broadcast chat messages to every other person on the grid. EspGrid chat is most typically used in improvisations to have discussions about the timing and nature of changes of direction, and in fixed compositions to indicate the timing of changes of section, special events, etc. Jokes, complaints, congratulations, and observations about the performance in progress are also common!

5. INTERFACE WITH PERFORMANCE ENVIRONMENTS AND APPLICATIONS

While some resources are engaged directly by a user working with the EspGrid GUI, a performer's experience usually involves working with what the public protocol sends to performance environments and applications. The application's Preferences allow the user to control whether the public protocol is sent to specific local applications or not, following a fixed, arbitrary scheme of UDP ports.

In addition to these "standard" public protocol connections, the application can be told to forward the public protocol to custom addresses and ports. This allows for flexible uses locally, as well as for the participation of machines that cannot run any available implementation of EspGrid (current implementations are for Mac OS X and iOS). An EspGrid instance becomes a "buddy" to the machine without EspGrid.

5.1.1. Public protocol sent by EspGrid

These are the OSC messages sent by EspGrid to other local applications as part of the public protocol:

- /esp/beat [n=beatNumber] [l=length of cycle] [d=duration of beat in seconds]
- /esp/chat [name-of-sender] [rest is message]

5.1.2. Public protocol received by EspGrid

Each instance of the grid currently responds to the following OSC messages, as an alternate means for controlling the grid when the GUI is impractical:

- /esp/beat/on [1 or 0]
- /esp/beat/tempo [beats per minute]
- /esp/beat/cycleLength [number of beats in bar]
- /esp/chat/send [name-of-sender] [rest is message]

5.1.3. Helper objects

An adjunct element of the project that is critical in routine use consists of a number of helper objects created to conveniently parse the EspGrid public protocol within ChucK and Max. Helper objects for other environments are necessary and not hard to create!

6. FUTURE DEVELOPMENT

Future development will maintain two key current features: the extremely minimal demands EspGrid places on the user and the small GUI footprint. As new features emerge, top priority will be given to keeping the public protocol as invariant as possible. High-level resources to be added in future versions of EspGrid include audio/video sharing, screen casting, and the ability to form grids across multiple local area networks.

7. ACKNOWLEDGEMENTS

This work was supported by the Social Sciences and Humanities Research Council (SSHRC) of Canada, and the Arts Research Board of McMaster University.

8. REFERENCES

[1] Trueman, D. (2007). "Why a laptop orchestra?" *Organised Sound* 12(2): 171-9.

[2] Ogborn, D. (2012). "Composing for a Networked, Pulse-Based, Laptop Orchestra." *Organised Sound* 17(1): 56-61.

[3] Collins, N., McLean, A., Rohrhuber, J., and Ward, A. (2003). "Live coding in laptop performance." *Organised Sound* 8(3): 321-30.

[4] Wang, G., Cook, P.R. (2003). "ChucK: A Concurrent, On-the-fly, Audio Programming Language." *Proceedings of the 2003 International Computer Music Conference.*

[5] Wessel, D., Wright, M. (2002). "Problems and prospects for intimate musical control of computers." *Computer Music Journal* 26(3): 11-22.

[6] Cristian, F. (1989). "Probabilistic clock synchronization." *Distributed Computing* 3(3): 146–58.

[7] Salazar, S., Wang, G., Cook, P.R. (2006). "miniAudicle and ChucK Shell: New Interfaces for ChucK Development and Performance." *Proceedings of the 2006 International Computer Music Conference.*