THE CARNEGIE MELLON LAPTOP ORCHESTRA[1]

Roger B. Dannenberg, Sofia Cavaco, Eugene Ang, Igor Avramovic, Barkin Aygun, Jinwook Baek, Eric Barndollar, Daniel Duterte, Jeffrey Grafton, Robert Hunter, Chris Jackson, Umpei Kurokawa, Daren Makuck, Timothy Mierzejewski, Michael Rivera, Dennis Torres, Apphia Yu
Carnegie Mellon University
School of Computer Science

ABSTRACT

The Carnegie Mellon Laptop Orchestra (CMLO) is a collection of computers that communicate through a wireless network and collaborate to generate music. The CMLO is the culmination of a course on Computer Music Systems and Information Processing, where students learn and apply techniques for audio and MIDI programming, real-time synchronization and scheduling, music representation, and music information retrieval.

1. INTRODUCTION

Starting with the League of Automatic Music Composers and the Hub [1], the idea of networked computers running semi-autonomously to compose and perform music has been a fascinating subject. Recently, the Princeton Laptop Orchestra [6] was created, with input from composers, computer music researchers, and students. Inspired by these efforts, we formed the Carnegie Mellon Laptop Orchestra, or CMLO, as part of a course offered by the Computer Science Department.

The design and musical directions of the CMLO are determined by the instructional goals of the course and by the musical knowledge and background of the students. The main goal of the class is to learn how to design and implement real-time interactive music systems (and related systems such as robotics, embedded systems, games, and process control) by implementing small projects. Since most students are not studying music, they come with a variety of backgrounds and experience. All are familiar with popular music of some kind, but few are well versed in contemporary art music. The CMLO is structured around very conventional music, but offers some innovation in the areas of modular software for music making, network control and delivery of music, and real-time interaction with compositional algorithms.

Reflecting the emphasis of the couse, we describe the software architecture of the CMLO, the workings of the components, and finally the outcome of a public concert. Overall, the CMLO was a great success, not only in terms of producing music but also in motivating students to integrate and apply what they learned in the course. Undoubtedly, working in a software development team, coordinating changing specifications, and integrating and testing software were all valuable additions to the course.

2. CMLO SYSTEM DESIGN

Class projects are difficult to design. On the one hand, it is good if everyone has a job to do that somehow supports the whole. On the other hand, it is risky if the overall success depends upon each component working perfectly. The CMLO has some critical components, but many are not so critical for the success of the whole system. In most cases, we assigned two students to work together on the most critical components.

2.1. Components

The overall design, as shown in Figure 1, is based on a central hub that relays messages to clients. Although the central hub must not fail, the system allows clients to join and leave the network on-the-fly. So, for example, a client can be stopped, the code can be edited, and the client can be restarted and re-attached to the network.

Most of the clients represent virtual musicians, serving conventional roles that include: drummer, bass player, chord player, melody player, and arpegiator. These musicians are controlled by information that is equivalent to a "lead sheet." In other words, musicians get time signature, tempo, key signature, and chord symbols, but no information at the note level. This gives the creator of the musicians the opportunity to explore any number of compositional algorithms and generative music techniques. Musicians are also created for different styles that include blues, funk, and techno. Some musicians can handle multiple styles, and some are specialists. The specialists simply stop playing if the current musical style is not within their capability.

[1]Originally published as: Roger B. Dannenberg, Sofia Cavaco, Eugene Ang, Igor Avramovic, Barkin Aygun, Jinwook Baek, Eric Barndollar, Daniel Duterte, Jeffrey Grafton, Robert Hunter, Chris Jackson, Umpei Kurokawa, Daren Makuck, Timothy Mierzejewski, Michael Rivera, Dennis Torres, and Apphia Yu. "The Carnegie Mellon Laptop Orchestra." In *Proceedings of the 2007 International Computer Music Conference, Volume II.* San Francisco: The International Computer Music Association, (August 2007), pp. II-340 - 343.

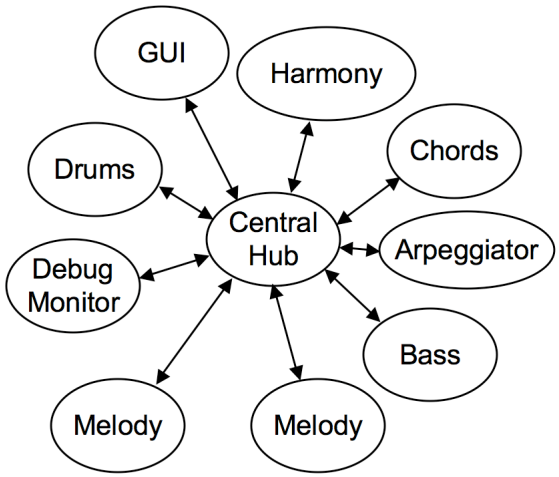

Figure 1. System organization of the CMU Laptop Orchestra.

The musicians are coordinated and controlled by sending them state information via the network. Some information, e.g. a chord progression, is generated algorithmically, and all the control information originates from a program called Harmony. This program makes a plan for a group of measures and sends this plan to all the musicians through the central hub.

Harmony, in turn, responds to some high-level controls that include tempo, key, and musical style. This information is manipulated using a program with a graphical interface. Like the other components the graphical interface can join the network dynamically. Figure 2 shows the interface used to control the orchestra.

For example, if a user decides to change the current style to Blues, he or she can choose this style in the GUI (Figure 2). As a result, the interface process sends a style-changing message via the central hub to the Harmony process. Once it receives that message, the Harmony process generates a new chord progression for that style and sends this chord progression along with style information to the virtual musician processes, again via the central hub. In order to have all musicians start the new style at the same time, the Harmony process adds a timestamp to each message and sends the messages well ahead of real time. When a musician receives the style information, it checks if it knows how to play this style, in which case it uses the chord progression information to generate music. The details of how style, chords, and other information are interpreted to improvise music are left up to the individual clients.

Some debugging support was written to help debug the complete system. The debug monitor program can join the network, monitor all commands coming from the Harmony program, and examine the behavior of the clock synchronization protocol.

Other modules were contemplated, including synthesis modules (currently musicians use local MIDI synthesizers) and modules to collect and stream MIDI to remote concert sites. As it turns out, we used Skype (www.skype.com) to transmit audio and video from the concert to the instructor, who was (safely) over 3000 miles away.

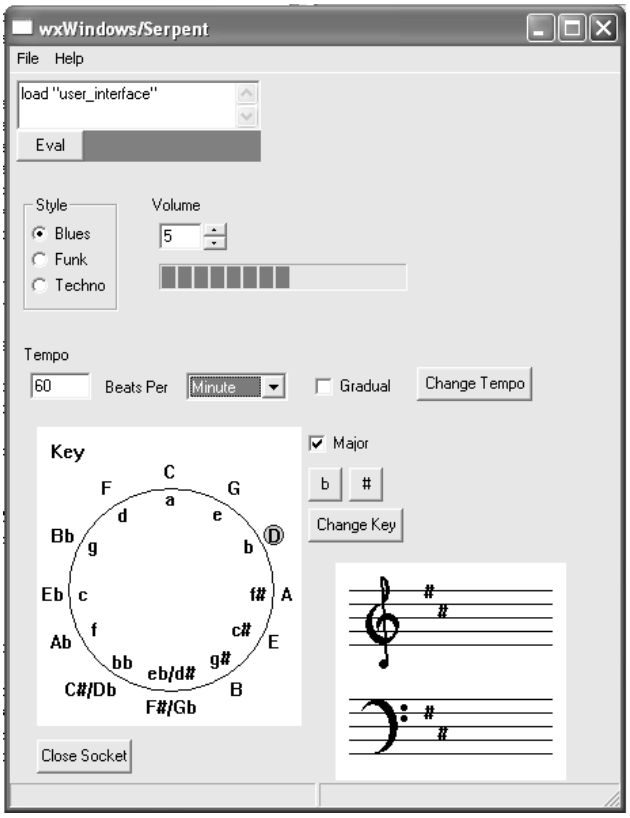

Figure 2. Graphical user interface to control style, volume, tempo, and key during performances.

2.2. Message Communication

The various processes communicate via TCP/IP through a central *hub* or server, and all other processes (musicians, Harmony, graphical interface, debugging tools) are called *clients*. Routing all messages through a central process has obvious drawbacks, but in practice, it greatly simplifies communication by: (1) establishing a single connection point for each component, (2) centralizing the recovery code needed for dropped connections (clients assume that the central hub never stops, and only the hub needs to handle the case where a client crashes or disconnects), (3) creating a single authority for time synchronization, and (4) providing a mechanism for locating desired recipients of messages.

Although OSC (http://opensoundcontrol.org) [7] is frequently used as the foundation for distributed music applications, our need for many-to-many communication led us to create our own protocol just for this distributed application. Messages are sent from clients to the hub,

which then forwards messages to appropriate clients. The messages are:

- *I_am*: used by a client when connecting to the hub. Identifies the client by category such as "drum," "bass," "ui," or "har;"
- *Style*: sets the style to "techno," "blues," *etc*.;
- *Time*: specifies the real time of a given beat;
- *Tempo*: specifies a tempo change;
- *Volume*: specifies volume scale factor;
- *Tchange*: requests a sudden tempo change;
- *Accel*: requests a gradual tempo change;
- *Key*: specifies a key change;
- *Chords*: specifies a chord as a pitch class set, a root, and the position (measure) of the chord within a sequence of chord changes.

For simplicity, messages are all strings of plain text. Messages are sent to the hub and optionally include a set of logical destinations. For example, the graphical interface program can send a *Tchange* message with a destination of "har." The hub will forward this message to the Harmony program (which identified itself to the hub using an "*I_am* har" message). The Harmony program will then plan a suitable time to introduce a tempo change (usually the end of an 8-bar phrase) and issue a *Tempo* message to all clients.

2.3. Synchronization

Synchronization in the laptop orchestra is important because of the distribution of machines, the potentially high latency of wireless communication, and the fact that the generated music is beat based. Synchronization is achieved in a layered fashion and relies heavily on time-stamped messages. The important principle here is that the client behavior should depend on the message *content* including timestamps, but not on the message *delivery time*, which is difficult to control.

Synchronization is achieved though several layers of timing specification. At the top layer, musical activities are scheduled according to integer beat numbers. For example, chord changes or style changes will take place on integer beat boundaries. Changes are sent in advance of the actual beat to allow time for messages to propagate, and when the messages arrive early, clients use a scheduler to delay the processing of the message until the indicated beat time.

To determine when beats occur, there is a system-wide mapping from real-time to beat-time. This mapping is specified as a simple linear function with two coefficients (offset and rate). This specification avoids any need for clients to keep a cumulative record of tempo changes in order to know the current beat, and therefore clients can join the network at any time and quickly synchronize with other clients.

At the lowest level of timing specification, all clients must have a shared view of the global real time. Elaborate schemes, including NTP [4] could be used, but for simplicity and pedagogical purposes, we embedded time synchronization into the clients and the central hub, which serves as a time server. The method is quite simple: clients periodically request the real time from the hub. When the time t_r returns, the client knows that this time was correct at some point between when the request was issued (local time t_1) and when the reply was received (local time t_2). If these values are sufficiently small, which is usually the case, the client estimates that the real time was correct at the half-way point between sending the request and receiving the reply. Thus the offset between hub time and local time is approximately $t_r - (t_2 - t_1)/2$. If the difference $(t_2 - t_1)$ is large, this round of synchronization is simply ignored and the previously estimated offset is used until the next round.

There was some concern that network traffic would either be too heavy or the delays would be unacceptable. The protocols were designed conservatively in anticipation of long delays. We also planned to have a backup hub running on a wired Ethernet, which would cut the wireless traffic approximately in half. In practice everything worked very well, and we even ran Skype with an audio and video connection over the same wireless network without problems.

2.4. Program Structure

The typical laptop orchestra program polls constantly for tasks to perform. Polling is accomplished either by a simple loop or by using a periodic function call. The polling function checks for an incoming message and calls a scheduler routine to test for any ready-to-run events. If a message arrives, it is interpreted. For example, a *Key* message, specifying a key change, normally schedules a key change at the beat indicated in the message. The scheduler maintains a sorted list (a priority queue) of events. Each event is represented by time (in beats), a function to be called, and parameters to be passed to the function [2]. For example, the *Key* message handler might schedule an event at beat 64 to call the function named *key_change*, with parameters indicating the key of d-minor. At beat 64, this event becomes ready to run, so the scheduler calls *key_change* and removes the event from the queue.

This scheduler is also used to generate music. For example, a simple bass player that plays quarter notes can be implemented as a function that looks at a chord progression data structure to find the current chord and select an appropriate pitch to play. After sending a MIDI key-down message to start the note, the bass player function schedules a key-up message one beat into the future. The bass player function also schedules an event to call back to the same bass player function one beat in the future. This generates an infinite sequence of quarter notes. Incoming *Chords* messages will change the chord progression data structure, affecting pitch selection.

Of course, this simple scheme can be elaborated. The bass player might stop (by not rescheduling itself) based on the current style, and the message handler for *Style* might need to start the appropriate bass player. The bass player can vary the rhythm by rescheduling itself at any beat time (or fraction thereof). For example, the current drummer algorithm works by reading rhythmic patterns from arrays of rhythm data.

3. IMPLEMENTATION

The CMLO is implemented using Serpent [3], a real-time scripting language. Serpent was used because, as a scripting language, it hides many system-dependent data types, calling conventions, and details. In general, a Serpent program is much shorter than the corresponding Java or C++ program, but Java offers many more on-line resources and code examples. Most students with Java experience feel that a Java implementation would have taken about equal effort. Serpent has the advantage of a real-time garbage collector, but for this project, occasional brief stalls due to garbage collection would probably not be noticed.

Since early course assignments required students to implement and use their own schedulers and other useful components in anticipation of the CMLO project, it was believed that their existing software could be reused in the CMLO. However, one of the most limiting factors in the CMLO development was the constantly changing protocol specifications, and the coordination effort was almost overwhelming. In retrospect, it probably would have been better to develop some solid core components, including a scheduler, time synchronization module, message marshalling and parsing, and data structures for chord progressions and other music state information. This would have insured greater compatibility between clients and simplified system integration.

Another problem was that testing was difficult without all the system components in place. Leaving each student to construct their own test scaffolding creates much duplicated effort and in practice minimized the amount of testing students could accomplish. A ready-made test environment would have helped.

Pedagogically, the difficulties we encountered can be viewed as very valuable learning experiences. Also, leaving even the lowest levels of system protocol design to the students seemed to create a greater sense of ownership, interest, and ultimately pride in the resulting system. On the other hand, it would have been fun to spend more time on music generation and control, which might have been possible if more infrastructure already had been in place. The best path is perhaps a matter of choosing which lessons are most valuable.

The public performance of the CMLO took place in December, 2006. Laptops and portable speakers were arranged around an atrium that was simultaneously housing a poster session where students in other courses were presenting their work.

4. CONCLUSIONS

The CMU Laptop Orchestra provided a fun and challenging project as the culmination of a computer science course on computer music systems implementation. The orchestra provided system design and implementation challenges, a creative musical outlet, and an interesting platform for collaborative distributed music making. The platform serves to synchronize distributed virtual musicians, providing them with beats, tempo, chord progressions, and style, and the virtual musicians play familiar musical roles of bass, drums, melody, and harmony.

The potential for widespread distribution is quite interesting. We were almost able to install a graphical interface in Belfast during the performance in Pittsburgh. The potential is certainly there to locate conducting and control anywhere in the world. Any number of musicians can join the "jam session," and a future version might even support live musicians playing along with virtual ones. In this case, it might be necessary to run the virtual musicians well ahead of real time and deliver music early to the live players to avoid network latency.

Another potential is to create virtual bands for virtual worlds such as Second Life. Here, real and virtual players might gather to perform for virtual and real audiences. The Improv system [5] provides an example of how these virtual players might be created. Perhaps a virtual version of "American Idol" would provide an interesting showcase for music generation algorithms and computer music in general.

REFERENCES

[1] Chadabe, J. *Electric Sound: The Past and Promise of Electronic Music*. Upper Saddle River, New Jersey: Prentice Hall, (1997), pp. 295-297.

[2] Dannenberg, "Software Design for Interactive Multimedia Performance," *Interface - Journal of New Music Research*, 22(3) (August 1993), pp. 213-228.

[3] Dannenberg. "A Language for Interactive Audio Applications," in *Proceedings of the 2002 International Computer Music Conference*. San Francisco: International Computer Music Association, (2002), pp. 509-15.

[4] Mills, D. *Network Time Protocol (NTP)*. IETF RFC 958.

[5] Singer, E., Goldberg, A., Perlin, K., Castiglia, C., and Liao, S. “Improv: Interactive Improvisational Animation and Music” In *ISEA 96 Proceedings, Seventh International Symposium on Electronic Art*. Rotterdam, Netherlands: ISEA96 Foundation (1997).

[6] Trueman, D., Cook, P., Smallwood, S., and Wang, G. “PLOrk: Princeton Laptop Orchestra, Year 1” In *Proceedings of the 2006 International Computer Music Conference (ICMC)*, New Orleans, U.S., November 2006.

[7] Wright, M., Freed, A., "Open Sound Control: A New Protocol for Communicating with Sound Synthesizers", in *International Computer Music Conference*, Thessaloniki, Greece, 1997.