CNMAT INFORMATION INFRASTRUCTURE

Andrew Schmeder, Matthew Wright, Adrian Freed, Edmund Campion, and David Wessel
Center for New Music and Audio Technologies (CNMAT), Department of Music
University of California (UC) Berkeley, Berkeley, CA 94720
{adrian, andy, wessel, matt}@cnmat.berkeley.edu

ABSTRACT

We describe the new unified information infrastructure CNMAT uses for organizing and publishing its music production, teaching, research and development and administrative activities with a focus on integration of collaboration, semantic web, and version control technologies.

1. INTRODUCTION

Activities at CNMAT continue to grow at a healthy, rapid pace, including public performances, research projects and publications, classes, and student research and composition. One major change has been a newly formalized relationship between CNMAT and the larger Department of Music: CNMAT is now committed to supporting all aspects of the expanding Music and Technology curriculum in the Department, including the new Music and Technology requirement for music composition graduate students. We continue to coordinate collaborative projects with many other UC Berkeley departments, including mathematics, statistics, mechanical engineering computer science, electrical engineering, psychology physics, and space sciences. We have increased the involvement of students in the mission-critical elements of music, research, and lesson production in what is referred to these days as "outcome-based" learning. Finally, the collection of CNMAT's technologies, teaching materials, and musical repertoire continues to grow [3], and we want to share this content worldwide.

We have addressed our need to grow with a moderate staff increase – one person to focus on pedagogy – and with the development and deployment of a new unified information infrastructure to manage our internal operations, campus and sponsor co-ordinations, public outreach and communications.

This report will focus on unusual and novel features of this information platform and how we use it to empower individuals to collaborate well, efficiently complete tasks with minimal face-to-face co-ordinations and avoid duplication of effort.

2. ACTIVITY MODEL

We are able to employ a unified information architecture because we have observed that the diverse workflows we wish to facilitate all fall within the design pattern "Explore-Coordinate-Create-Publish." In our activity model the details of specific goals, permissions, and required resources vary greatly among activities but the user interface and underlying conceptual and computational machinery to support the activities is the same. In our small multidisciplinary center each user performs many different activities, so it is economical to have a shared user interface to reduce the cost of infrastructure support as well as the memory burden on users.

We illustrate our model with two contrasting activities: coordinating a public concert and publishing a revision of a Max/MSP external. The *exploration* stage of the former involves a search within the CNMAT information infrastructure and external sources for possible dates, performers, venues, and technical resources. The *coordination* phase involves the reservation of the identified resources required for the event and the approval of these reservations by the appropriate parties. Our reservation system combines an inventory system, a calendar, and a set of rules that allocate the inventoried resources. Inventoried items include entire rooms, microphones, books, software licenses, and musical instruments. The rule system is general enough to accommodate, for example, permanently assigning a software license to an individual user, or temporarily loaning a microphone to an event outside our building. A key to this system is the assignment of unique ID's to physical and virtual objects and unique identities for users so we can manage the appropriate permissions structure and access control. Once the resources are assigned, the event announcement can be created and, once approved, will be automatically *published* on our public calendar and e-mail list. After the concert takes place (the *create* step), all this information becomes part of an archive of the event, which might grow to include audio and/or video recordings of the performance, press and critical feedback about the event, and even an online forum where audience members and performers can discuss the event.

The software revision activity starts with an *exploratory* phase to identify the software component and (optionally) view its development history. The *coordination* phase consists of checking out the latest source code and any required development tools from inventory. A new version will then be *created* and checked into inventory. After testing and approvals the *published* view of the new software will be made available for download and subscribed users will be notified.

3. CONTENT MANAGEMENT: DRUPAL

3.1. Choice

There are dozens of content management systems to choose from, but, as we know from experience with other platform technologies, only a handful will survive in the long term. Migration in the future from orphaned technologies would be too expensive and is out of the question for us. After eliminating well-known blogging and bulletin-board platforms that are migrating slowly

into content management systems we chose Drupal[1] because:

- It now has critical mass to survive in the long term
- It is professionally implemented with quality control of outside contributions
- Drupal source is open, and Drupal's well-defined module architecture make it easy to adapt to our needs
- It was designed with a small number of powerful unifying constructs that facilitate our broad use of the platform. In particular we take advantage of its pervasive powerful taxonomy system and node customization.
- Many useful contributed modules are available, e.g., Bibliography, a publications database with EndNote and LaTeX compatibility

Since the beginning of the Internet we have experienced a rapid evolution of new interaction paradigms including static web pages, categorized links, dynamic web pages, blogs, bulletin boards, RSS feeds, search portals, etc. An attractive feature of Drupal is the complete separation of the node storage mechanism (the content) from the user interaction (the form). This allows us to adopt new content formats, network protocols and interaction paradigms as they become popular, without having to rewrite or reorganize content. For example we recently enabled modules that transparently exploit new efficient user interfaces based on AJAX.[2]

3.2. Nodes

The basic unit of information in Drupal is a *node*. Each community of users on the site is a *role*, e.g., the anonymous general public, students, staff, sponsors, etc. Access controls for each kind of node give permission to see, create, revise, organize, and publish on a per-role basis, encouraging collaborative development of content, e.g., a student adding a book recommendation. Each node (like a Wiki page) also has a revision history, including authorship and log messages. Nodes have titles, a summary, body text, and comments. Multiple document types and text formats are handled by input filters and node extensions, and output templates deliver the content in multiple formats including HTML, RSS, and printer-friendly HTML or PDF. Media node types are simply regular nodes with additional metadata fields and a link to media files. A *max patch* node contains a screen shot, links to authors, dependency list, and SVN repository path.

3.3. Taxonomy, vocabularies, terms, and tagging

As the number of nodes grows, the process of search and discovery becomes an increasingly critical component of the user experience. In many cases keyword searching can provide an efficient path for finding relevant items, but for popular keywords there may be too many search results. For example, it may be more useful to find all nodes about compositions by David Wessel rather than the much larger set of nodes containing the text "David Wessel." The Drupal taxonomy system enables us to evolve a comprehensive system of standard terminology for annotating the context of each node, a process that is called *tagging*.

The taxonomy consists of a set of vocabularies that are essentially the most concise possible set of terms that span the information infrastructure. Each vocabulary consists of a hierarchy of terms, each of which can be used for tagging a node.

Thus, each node is tagged with any number of terms drawn from vocabularies relevant to the node type. So instead of a single hierarchical segregation of all content into, e.g., *demos*, *repertoire*, *videos*, *course downloads*, etc., each node has enough tags to be included in multiple hierarchies. For example the node for the piece *Corail* has tags indicating that it is a musical composition, it uses pitch detection, it was written by Edmund Campion, it is for saxophone, it is an interactive real-time piece, it is part of the MMJ Depot [3], it uses pitch detection, live sampling, and multichannel sound spatialization, it was written in 2000-2001, etc. Each of these tags is a link to a page listing all of the nodes with the given tag, so for example someone interested to learn that *Corail* uses pitch detection could follow that tag to find other pieces using that technology, pitch detection software downloads, references to papers on the topic, etc.

Vocabulary design turns out to be one of the most challenging aspects of our information architecture [2]. Poorly tagged nodes will not be found or seen, and without careful design node authors will not understand how to tag effectively. Our approach to creating new entries in the taxonomy has been conducted in parallel with the data entry process for creating the detailed inventory of CNMAT's physical and intellectual assets. From this we derived a set of guidelines governing the management of the taxonomy as follows:

- *Independence:* Each vocabulary should be maximally independent of the others. We initially prefer terms that do not overlap in meaning, but eventually permit overlapping terms if they are defined properly as the summation of sets of other, more fundamental terms (accomplished in Drupal via "related terms"). Some terms unavoidably have multiple meanings (e.g. CNMAT as a *Venue* versus CNMAT as an *Organization*). In this case the term is duplicated in multiple vocabularies and the specific meaning is disambiguated by context.
- *Relevance:* A focus on our own internal operational terminology [3] is beneficial for users and keeps the task tractable (i.e., we do not attempt to describe the universe as a whole).
- *Minimum Structural Overhead:* Only admit hierarchy where a clear class-subclass relationship exists, i.e., the parent term can properly be described as being the same type of entity as the child terms. (This is called an "is-a" relationship.)

3.4. Vocabulary examples

Taxonomy vocabularies are typically focused on topics or terminology, and here we draw upon useful studies

[1] http://www.drupal.org
[2] www.adaptivepath.com/publications/essays/archives/000385.php
See also http://www.w3.org/TR/XMLHttpRequest

of computer music terminology [1]. However our taxonomy must encompass our entire information infrastructure and therefore include many attributes that cannot be included in a subject taxonomy. For illustration a selection of vocabularies is presented here:

- "Equipment", "Software", "Locations" and "Standards" are used for tagging the nodes that make up our inventory system.
- "Venues", "Works", "Projects", and "Events" contain terms that can describe much of the activity around the presentation of music.
- Some vocabularies cut across the information infrastructure by their broad applicability including "Organizations", "People", "Roles", "Standards", and "Audience".

4. VERSION CONTROL: SUBVERSION

4.1. Choice

We chose Subversion[3] (SVN) for version control of our source code (including Max/MSP patches) and associated files such as examples, documentation, external libraries, SDKs, etc. We would have liked to use a single system for all content management, but Drupal currently does not meet our needs for these kinds of files.[4] Our solution is to use Drupal to coordinate software projects, track bugs, and expose our downloadable software on web pages, to use SVN for revision control, and to automate the software release process with custom bridging software. The following features make SVN attractive:

- SVN keeps track of the history of revisions of each file. SVN's model is that the repository as a whole has a revision number, so you can always go back to the complete state of all software in the repository as of a certain time in the past or corresponding to a particular version of a single file.
- SVN maintains the identity of a file (i.e., its history) even if the name or location changes (unlike CVS). This makes it possible to check in a large tangled mess of software (thereby preserving it) and then later go back and reorganize it.
- SVN can easily show the difference between any pair of versions of a text file.
- SVN allows for external authentication (i.e., Drupal users).
- The repository exists in a known central location, so we always know where to find the current version of any software (that has been checked in).

[3] http://subversion.tigris.org

[4] Although Drupal maintains the history of revisions of each node, its relationship module is not mature enough to manage dependencies between content. Although Drupal can detect the case when somebody changes a node and then somebody else tries to submit an update of that node that was based on the earlier version, it currently provides no support for resolving this conflict, and simply rejects the second submission. Drupal has no concept of a working copy. Finally, many existing software development tools interact with specialized version control systems such as CVS or SVN.

4.2. Workflow and terminology

First a collection of files is *checked out* of a *repository* to create a local *working copy* where they may be edited and run. The *status* of a directory in a working copy lists which files have been modified, added, deleted, etc. A *diff* of a file shows what has changed from the version checked out, or, optionally, shows the changes between any pair of versions in the repository. We can *add*, *delete*, and *rename* files in a working copy; the *commit* operation then stores these and any other changes in the repository.

4.3. Subversion clients

Some programmers comfortable with the Unix shell use the standard command-line *svn* client. Others prefer a GUI that more resembles the Macintosh Finder, i.e., *svnX*. Also the text editor *BBEdit* performs some basic Subversion operations. On Windows, the *TortiseSVN* client integrates with Windows Explorer (e.g., adding items to the control-click contextual menu); the command-line SVN client also runs on Windows under Cygwin.

The biggest burden in using Subversion is that all directory operations (especially moving, renaming, and removing files) must be done via a Subversion client so that the repository can keep track of the history. Thus a user can't reorganize a directory structure just by moving the files around normally.[5]

4.4. Repository layout

A Subversion repository is a large nested directory structure; a working copy is a subset of this structure. The design of our directory layout

- Makes it easy to find what's already in the repository.
- Makes it easy to choose "the right place" for new items to be added to the repository.
- Promotes factoring and helps minimize version skew, e.g., by putting libraries for OpenSoundControl and SDIF in a single place and making projects that use them (e.g., Max externals, TASS, SDIF command-line utilities, dumpOSC, sendOSC…) refer to the libraries with relative paths within the repository.
- Provides personal "sandboxes" for users to store versions of their unfinished work.

5. DOWNLOADS

5.1. Software information and versioning system

Each unit of software has an associated name/value table giving its name, description, author(s), copyright years,

[5] Other version control systems, notably Linus Torvalds' *git* (git.or.cz) instead use heuristics to guess which files are derived from which other files. ("Git tracks content not files.") He makes some good points to justify this, such as the difficulty in representing cases where two files are merged to create a single file (http://marc.info/?l=git&m=114123702826251). Unfortunately no such system is sufficiently integrated with our existing software development tools for us to move beyond Subversion at this time.

version history, Drupal tags etc. For software written in a textual language (including Max/MSP externals written in C, Java, or Javascript), this table is embedded in a specially formatted comment inside the main source code file; for other software, e.g., a collection of Max patches, this table lives in a separate text file. Our system automatically propagates this information to the version number on the downloads page and, for Max/MSP externals, to the message that prints when the external first loads, the output of the external's "version" method, etc.

5.2. Release process

Releasing something for public or internal download is different from checking it into the repository. We use SVN during software development to record the history of successive refinements and to facilitate collaboration; only after software lacks obvious bugs and meets our standards of usefulness and clarity will we release it.

The entire build/release process is controlled with makefiles.[6] For example for Max/MSP externals written in C there are separate "make" targets for each platform, e.g., "make macho," "make win," etc., as well as separate targets to release a new version, e.g., "make macho-release." The makefile invokes xcodebuild, Codewarrior, or gcc/mingw depending on the platform, then packages the resulting binary external with its help file and any other files that need to be distributed with it; the name of this archive file includes the version number and platform. Prior to the compilation is a step where the external's name, description, author(s), version number, etc., are parsed (via Perl scripts) as described above and reformatted as #define statements in a file called *version.h* so that the symbols NAME, DESCRIPTION, VERSION, etc., in the C code will have the right values.

The release process first makes sure that the versions of all the files that go into the release are checked into SVN (thereby guaranteeing that we can always go back to the source code for any released software version), and also that each text file uses the proper style of line break character. The Makefile then automatically creates an appropriately named download archive and an associated name/value table text file. The name/value table contains taxonomy tags and other metadata required by Drupal to synthesize the appropriate download web pages. Our system also generates "everything" archives for a specific topic (e.g., all Max/MSP externals, all CNMAT downloads having to do with *Timbre Space*, all software to run a specific piece) on multiple platforms.

6. CONCLUSIONS AND FUTURE WORK

The new CNMAT information architecture is a strong foundation for our future interaction with the computer music and other communities, and we have already observed the early signs of increases in engagement and productivity. We expect that as these stronger information management platforms become more widespread, the community will be in a better position to solve some of the larger problems in the field such as a public version control repository to hold successive revisions of software and documentation for computer music repertoire. Particularly for pieces with an interactive real-time component, nearly every performance requires some kind of update to the software to make it work on modern hardware, operating systems, versions of Max, etc.; these changes should then be checked in so that the next performance can start from the latest (or any) version of the software.

Subversion's *diff* capabilities are extremely useful for tracking software's revision history, but unfortunately not very helpful for Max patches. For patches stored in binary format, or any other binary file (images, sound files, movies, pdf files…), all *diff* can say is whether two versions are identical or different. Max also has a text patch format, but viewing diffs of these generally provides no insight: a small change to a Max patch can result in changes in a very large number of lines of the text format version of the patch. What's needed is a sophisticated tool for visually highlighting patch differences; this should be a lot easier with the forthcoming Max 5.0 patcher format. Increased patch structure parsing will also enable each *Max patch* node to link automatically to all the other patches that use it.

UC Berkeley already assigns unique campus ID's to each user for various administrative services such as e-mail, course enrolment, and library privileges. We plan to authenticate against the same to avoid users having extra ID's and to leverage the campus support for automatically managing the enrolment and departure of ID holders.

The Drupal taxonomy system currently supports only the "is-a" relationship between terms, but this is being generalized. We look forward to creating richer node relationships when stronger ontology support is available.

We will be eventually syndicating and aggregating more content from the broader community as we complete the formidable ongoing task of building nodes that represent our own legacy.

7. ACKNOWLEDGEMENTS

Edmund Campion, Jeremy Hunt, Jeff Lubow, Jen Wang, Derek Wright, Michael Zbyszynski, David Zicarelli. UC Berkeley Department of Music.

8. BIBLIOGRAPHY

[1] Pope, S. T. "A taxonomy of computer music", *Contemporary Music Review*, 13:2, pp. 137–145, 1996.

[2] Svenonius, Elaine. *The Intellectual Foundation of Information Organization*. MIT Press, Cambridge, MA, 2000.

[3] Zbyszynski, M, M. Wright, and E. Campion, "Design and Implementation of CNMAT's Pedagogical Software", *Proceedings of the International Computer Music Conference,* Copenhagen, Denmark, 2007.

[6] For Windows builds, even "project settings" live in makefiles, for example, the list of additional source code (e.g.,SDIF library) needs to be linked in. For Macintosh builds these data unfortunately still live in project files specific to the Xcode and Codewarrior IDEs.