Auracle: a voice-controlled, networked sound instrument*

JASON FREEMAN, KRISTJAN VARNIK, C. RAMAKRISHNAN, MAX NEUHAUS, PHIL BURK and DAVID BIRCHFIELD

College of Architecture, Music Department, Georgia Institute of Technology, 840 McMillan Street, Atlanta, GA 30332-0456, USA
E-mail: jason@jasonfreeman.net
Akademie Schloss Solitude, Solitude 3, D-70197 Stuttgart, Germany
E-mail: kristjan.varnik@akademie-solitude.de
Institut fuer Musik und Akustik, ZKM, Lorenzstraße 19, D-76135 Karlsruhe, Germany
E-mail: cramakrishnan@acm.org
350 Fifth Avenue, Suite 3304, New York, NY 10118, USA
E-mail: neuhaus@max-neuhaus.info
Softsynth, 75 Pleasant Lane, San Rafael, CA 94901, USA
URL: http://www.softsynth.com/contacts.html
Arts, Media, and Engineering, Arizona State University, Tempe, AZ 85281, USA
E-Mail: dbirchfield@asu.edu

Auracle is a voice-controlled, networked sound instrument that enables users to control a software synthesizer with their voice and to interact with each other in real time over the Internet. This paper discusses the historical background of the project, beginning with Neuhaus' 'virtual aural spaces' in the 1960s and relating them to Barbosa's conception of 'shared sonic environments'. The architecture of the system is described in detail, including the multi-level analysis of vocal input, the communication of that analysis data across the network, and the mapping of that data onto a software synthesizer.

Not only is Auracle itself a collaborative, networked instrument, but it was developed through a collaborative, networked process. The project's development mechanisms are examined, including the use of existing tools for distributed development, the creation of custom development applications, the adoption of extreme programming practices, and the use of Auracle itself as a means for communication and collaboration among developers.

1. INTRODUCTION

Auracle is a voice-controlled, networked sound instrument conceived by Max Neuhaus and realised collaboratively by the authors. Users interact with each other in real time over the Internet, playing synthesised instruments together in a group 'jam'. Each instrument is entirely controlled by a user's voice, taking advantage of the sophisticated vocal control that people naturally develop learning to speak. The project was designed to facilitate the kinds of communal sound dialogue that are rare in contemporary society:

> Anthropologists in looking at societies which have not yet had contact with modern man have often found whole communities making music together. Not one small group making music for the others to listen to, but music as a sound dialogue between all the members of the community . . . these works are really . . . proposing to reinstate a kind of music which we have forgotten about and which is perhaps the original of the impulse for music in man. Not making a musical product to be listened to, but forming a dialogue, a dialogue without language, a sound dialogue.
>
> These pieces then are about taking ordinary people and somehow putting them in a situation where they can start this nonverbal dialogue. They have the innate skills as our ability with language demonstrates. The real problem then is finding a way to let them escape from their preconceptions of what music is. We now think of music as an aesthetic product. When you propose to a lay public that they make music together, they all try to imitate professional musicians making a musical product, badly. It only gets interesting when they lose their self-consciousness and become themselves. (Neuhaus 1994)

Auracle is an entity made to create such an opportunity. It was designed to be accessible to a lay public without musical training or technical expertise. We strived to create an open-ended architecture rather than a musical composition: a system that, as much as possible, responds to but does not direct the activities of its users. We also sought to build a highly transparent system, in which users could easily identify their own contributions within the ensemble, while also remaining engaged over extended periods of time.

2. HISTORICAL BACKGROUND

Auracle is inspired by a class of analogue sound works that construct virtual aural spaces using networks comprised of telephone connections and radio broadcasts. In *Public Supply* (1966, 1973) and *Radio Net* (1977), Max Neuhaus mixed together audio from

*The Auracle project is a production of Max Neuhaus and Akademie Schloss Solitude (art, science and business program) with the financial support from the Landesstiftung Baden-Würtemburg. We express our gratitude for their generous support. Auracle is available at http://auracle.org

callers during live radio broadcasts and used callers' audio to control sound synthesis (Neuhaus 1990, 1994). More recently, radio shows by NegativLand (Joyce 2005) and Press the Button (Radio Show Calling Tips 2005), among others, have invited telephone callers to join improvising musicians in the broadcast studio.

These analogue networks are precursors to a number of digital works which use the Internet to create virtual aural spaces. Barbosa labels such works 'shared sonic environments' and defines them as 'a new class of emerging applications that explore the Internet's distributed and shared nature [and] are addressed to broad audiences' (Barbosa 2003: 58). As examples, he cites *WebDrum* (Burk 1999), where online participants collaboratively alter settings on a drum machine; *MP3Q* (Tanaka 2000), where users collectively manipulate MP3 files with a 3D interface; and *Public Sound Objects* (Barbosa and Kaltenbrunner 2002), an open-ended architecture for the creation of shared sonic environments.

We consider Auracle to be a shared sonic environment, and our work was influenced by Barbosa's examples as well as other recent projects. In *Daisy-Phone* (Bryan-Kinns and Healey 2004), for example, Internet or mobile-phone users collaboratively modify a looping musical MIDI sequence, with each user colouring circles to change the pitches and rhythms in his or her instrument. In *Eternal Music* (Brown 2003), each user drags a ball around a window to control a drone generated by modulated sine waves. Components of the *Cathedral Project* (Duckworth 2000), the *Brain Opera* (Machover 1996) and *HubRenga* (Brown and Bischoff 2003) have invited Internet users to control sounds during live physical performances, collaborating not only with other Internet users but also with live performers onstage. And *Silophone* (The User 2000) operates in both the analogue and digital domains; it joins together sounds made by telephone callers and sound files uploaded by Internet participants, playing them in a giant grain silo in Montreal and broadcasting their acoustic transformations back over the phone and Internet to participants.

Unlike *Silophone*, Auracle uses analysis data from the voice to control a synthesis engine; it does not directly process and output an audio stream. This approach was initially motivated by practical considerations: it reduced network bandwidth and latency while maintaining high-quality audio output. It is also a much more flexible way for us to map input to output.

Similar ideas about vocal analysis and synthesis can be traced back to the vocoder (Dudley 1939), which analyses voice input with a bank of bandpass filters and resynthesises an approximation of the original signal (Roads 1996: 197–8). And a number of recent software projects and interactive musical works use related techniques. For example, the Kantos software plug-in (Antares 2004) maps pitch, rhythmic and formant analyses of a monophonic audio input onto its synthesizer; the mapping parameters and synthesis algorithm are configured through a graphical interface. In *The Singing Tree* (Oliver 1997), a component of the interactive 'Mind Forest' in Tod Machover's *Brain Opera* (Machover 1996), users are asked to sing a steady pitch into a microphone; as they hold it steadier and longer, a MIDI harmonisation becomes richer, and images on a screen begin to change. And the *Universal Whistling Machine* (Böhlen and Rinker 2004) analyses the pitch and amplitude envelopes of a user's whistling, and it synthesises responses in which the tempo, contour and direction of the analysis data are transformed. Analogue synthesis based on vocal analysis was also an important component in Neuhaus' *Public Supply III* (1973) and in *Radio Net* (Neuhaus 1990 and 1994).

3. ARCHITECTURE

As the video example on the accompanying DVD demonstrates, users launch Auracle from the project's website, opening a graphical user interface through which they can 'jam' with other users logged in from around the world. To control their instrument, users input vocal gestures into a microphone. Their gestures are analysed, reduced into control data, and sent to a central server. The server broadcasts that data back to all participating users within their ensemble. Each client computer receives the data and uses it to control a software synthesizer.

The client software is implemented as a Java applet incorporating the JSyn plug-in (Burk 1998), and real-time collaboration is handled by a server running TransJam (Burk 2000). Data logging for debugging, usage analysis, and long-term system adaptation is

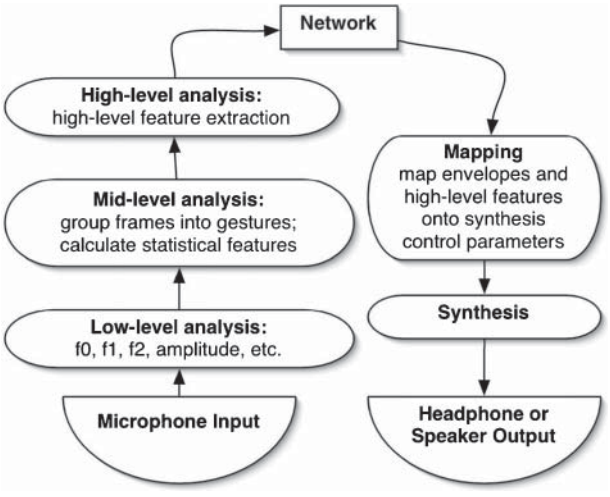

Figure 1. Auracle system architecture.

handled by an HTTP post (from Java) on the client side and PHP/MySQL scripts on the server side.

The following subsections describe each architectural component in detail.

3.1. Low-level analysis

The initial low-level analysis of the voice computes basic features of the audio signal over an analysis window, supplying data for the system to use in higher levels of analysis and in sound synthesis (Ramakrishnan, Freeman, Varnik, Birchfield, Burk and Neuhaus 2004). The analysis assumes that all incoming sound is vocal, because Auracle asks its users to make vocal sounds.

The incoming sound is analysed for fundamental frequency and root mean square (RMS) amplitude. We also detect features traditionally important to both speech and timbre analysis applications: the first two formant frequencies and their respective bandwidths, and the degree to which the sound is voiced or unvoiced. Voiced inputs are pitched sounds such as vowels, in which the vocal cords buzz; unvoiced inputs are breathier, noisier sounds such as fricatives and sibilants (Roads 1996: 204).

JSyn is used to capture the input from a microphone, but it cannot extract the vocal parameters we need, so we built this functionality ourselves. We limited our own DSP implementation to pure Java to avoid packaging and deploying native libraries for each targeted platform. We considered techniques based on linear prediction (LP), cepstrum (used in Oliver 1997), FFT, and zero-crossing counts. We chose linear prediction, feeling it would be the easiest to implement in pure Java with acceptable performance and accuracy.

Raw sample data from the microphone is brought from JSyn into Java. Once in Java, the data is determined to be voiced or unvoiced based on the zero-crossing count. Following Rabiner and Schafer (1978), the data is down-sampled to 8,192 Hz and broken into 40 ms blocks, which are analysed by LP for fundamental frequency, the first and second formant frequencies, and the formant bandwidths. The values for each block of analysis are fed into a median smoothing filter (Rabiner and Schafer 1978: 158–61) to produce the low-level feature values for that analysis frame.

Performance of the LP code was a major concern of ours. So, in this case, we violated Knuth's maxim and prematurely optimised. The LP code is implemented in a slightly peculiar, non-object-oriented style. The goal was to minimise virtual and interface method lookup, and more importantly, to minimise object creation. Though such issues are often disregarded when writing Java, removing memory allocations in time-critical loops proved crucial to tuning this code.

3.2. Mid-level analysis

The low-level analysis provides enough data to control aspects of the frequency, amplitude and timbre of a synthesizer with some degree of creativity, subtlety and musicality. But in practice, we found that a consistent one-to-one mapping of this low-level data did not allow participants to create a broad enough range of sounds. Without a higher level of transformation, the system failed to sustain their interest for extended periods of time. In response, we developed additional levels of analysis to classify vocal input. The system uses those classifications to further control the timbre of its synthesised responses.

The mid-level analysis groups low-level frames into gestures, so that further analysis can consider changes to low-level data over the course of these gestures. Since users are asked to hold down a play button while they are making a sound, it is trivial to create gestures based on the button's press and release. (Our original motivation for the button had been to reduce acoustical feedback in the system.)

Once Auracle identifies a gesture, it calculates a feature vector of statistical parameters which describe the low-level data. These statistics are based largely on studies of vocal signal analysis for emotion classification by Banse and Scherer (1996), Yacoub, Simske, Lin, and Burns (2003), and Cowie, Douglas-Cowie, Tsapatsoulis, Votsis, Kollias, Fellenz and Taylor (2001). While we are not focused solely on emotion, we found this research a useful starting point. Studies of timbre, most of which extend Grey's (1977) multidimensional scaling studies, were also informative, but their focus on steady instrumental tones was less directly applicable to the variety of vocal gestures expected from Auracle users. And while many emotion classification studies try to separate linguistically determined features from emotionally determined features (Cowie *et al.* 2001), we wanted Auracle to consider features of user input whether they were linguistically determined, emotionally determined, or consciously manipulated by users.

The mid-level feature vector includes forty-three features: the mean, minimum, maximum, and standard deviation of f0, f1, f2, and RMS amplitude, as well as of their derivatives; the mean, minimum, maximum, and standard deviation of the durations of individual silent and non-silent segments within the gesture; and the ratio of silent to non-silent frames, voiced to unvoiced frames, and mean silent to mean non-silent segment duration.

3.3. High-level analysis

The mid-level feature vector provides important analytical data about vocal input, but in a sense it provides too much data to be directly useful. How

could we reasonably map forty-three different statistical features onto synthesis control parameters?

Our solution is to perform a high-level analysis which projects the forty-three-dimensional mid-level feature vectors onto a three-dimensional space (Freeman, Ramakrishnan, Varnik, Neuhaus, Burk and Birchfield 2004). In defining those dimensions, we did not wish to merely select a subset of the mid-level features, nor did we wish to manually create projection functions; these approaches would have driven users to interact according to our own preconceptions, and in doing so would have contradicted the goals of the project. We were instead attracted to Principal Components Analysis (PCA), because it preserves the greatest possible amount of variance in the original data set. In other words, the mid-level features which users themselves vary the most take on the greatest importance in the PCA projection. It facilitates a self-organising, user-driven approach.

But PCA creates a static projection; for Auracle, we wanted a dynamic technique which could perform both short-term adaptation – by changing over the course of a session to focus on the mid-level features varied most by a single user – and long-term adaptation, in which the classifier's initial state for each session slowly changes to concentrate on the mid-level features varied most by the entire Auracle user base. With an adaptive PCA system, users are able to create an increasing variety of synthesised sounds as the system becomes more accustomed to the types of vocal gestures they are making. Even though each user explores only a small portion of the input space, his or her gestures are eventually projected onto an extremely large portion of the output space.

An adaptive classifier does sacrifice a degree of transparency in its classifications: it is more difficult for users to relate their vocal gestures to sound output when the high-level feature classifications, and thus the mappings, are constantly changing. And it is impossible to interpret the meaning of high-level features during the design of mapping procedures, since their semantics change with adaptation. For us, though, transparency in this component of Auracle was less important than adaptability.

Our adaptive PCA implementation does not use classical PCA methods, which define the principal components of a set of feature vectors to be the eigenvectors of the covariance matrix of the set with the greatest eigenvalues. This strategy is awkward to adapt to a continuously expanding input set and computationally expensive to perform in real time in Java. Instead, we implement the Adaptive Principal Component EXtraction (APEX) model (Kung, Diamantaras and Taur 1994; Diamantaras and Kung 1996), which improves upon earlier neural networks proposed by Oja (1982), Sanger (1989), Rubner and

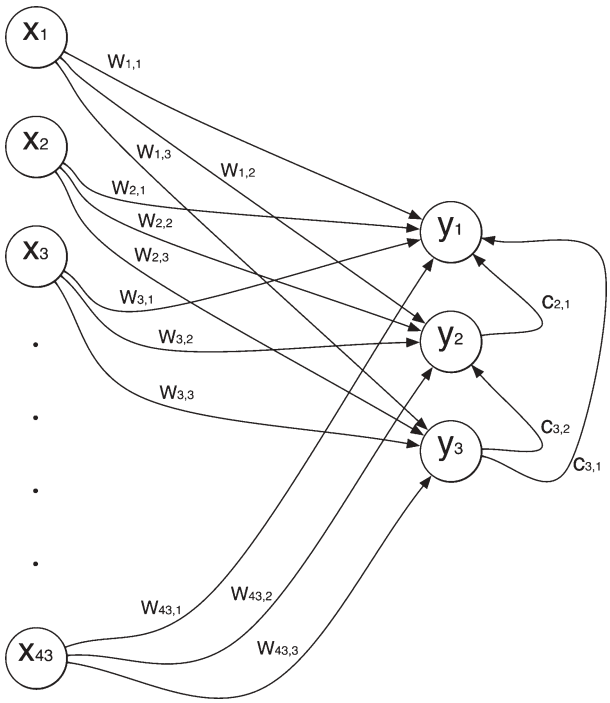

Figure 2. The APEX neural network as used within Auracle. X nodes represent mid-level features (input) and Y nodes represent high-level features (output). W weights are feed-forward, C weights are lateral.

Tavan (1989) and others. APEX efficiently implements an adaptive version of PCA as a feed-forward Hebbian network (with modifications to maintain stability) and a lateral, asymmetrical anti-Hebbian network. The Hebbian portion of the network discovers the principal components, while the anti-Hebbian portion rotates those components. The learning rate of the algorithm is automatically varied in proportion to the magnitude of the outputs and a 'forgetting' factor which controls the algorithm's memory of past inputs (Kung, Diamantaras and Taur 1994).

Upon launching Auracle, a client's neural network is initialised with weights from the server. (The initial server-side weights were created by training the neural network on a database of 230 recorded vocal gestures.) Over the course of a user session, the client-side neural network adapts to the vocal gestures created by the local user, updating its internal weights accordingly. Then, when a user logs out of Auracle, the client's internal weights are transmitted back to the server, which merges them with its previous weight matrix to facilitate long-term adaptation.

Unlike many other neural networks, it is easy to monitor how APEX adapts; each feed-forward weight represents the importance of a particular mid-level feature in the computation of a particular high-level feature. This transparency was critical in developing, debugging and evaluating the high-level analysis system within Auracle.

3.4. Network

The analytical techniques discussed in the previous sections do more than just analyse audio input; they are also an extreme form of data compression. Each Auracle client sends only a small amount of analysis data over the Internet instead of complete audio streams, dramatically reducing bandwidth requirements, network latency, and server load.

Each gesture's low-level analysis envelopes, along with the high-level feature values, are sent to a central server running TransJam (Burk 2000), a Java server for distributed music applications. TransJam provides a mechanism to create shared objects, acquire locks on those objects, and distribute notifications of changes to those objects. Each client stores its gesture data in a shared object, and when it updates that object, the server transmits the information to all clients in the ensemble. In this manner, all client machines maintain all players' analysis data in sync.

Java security restrictions and practical networking issues made direct peer-to-peer communication impossible, necessitating a central server. To mitigate the probability of a performance bottleneck, Auracle's architecture is designed to minimise the work done by the server. The server is merely a conduit for data and does no processing itself. Mapping and synthesis operations are duplicated by all clients, but we preferred this solution over increasing server load. Our benchmarking shows that we can support 100 simultaneous users, each sending one gesture per second, with an average CPU load of only 35 per cent on our Apple Xserve (1.33 GHz G4, 512 MB RAM).

3.4.1. Network timing and latency

Auracle is designed to facilitate a conversational style of interaction, in which players respond to earlier sounds they hear instead of planning simultaneous gestures with other participants. In online text chats and in most offline conversations, people listen to or read the comments of others before expressing their own thoughts; rarely do two people speak at the same time. Auracle encourages a similar approach, and in so doing, also circumvents many of the timing, synchronisation and latency problems common to networked music.

The analysis data is transmitted to the server only once a complete gesture has been detected. This reduces network traffic and generally uses the network more efficiently. Data is only mapped onto synthesis control parameters when it arrives from the server, even when the data was created by the local client. This creates a short delay between the vocal input and synthesised response which facilitates the conversational style of interaction.

In addition to the delays imposed by gesture detection and the latency of network transmission, the onset of a gesture is sometimes further postponed in order to minimise its overlap with other gestures. We delay gesture onsets by as much as one second, and when necessary we also time-scale gestures to as short as half their original length. When gestures are successive rather than simultaneous, the interaction becomes more conversational, and it also becomes easier to hear individual gestures within that conversation, increasing the system's transparency.

3.5. Mapping and synthesis

Once analysis data is received from the server, it must be transformed back into audio. A mapping component generates envelopes from the data and uses them to control parameters for a software synthesizer. Intuition, pragmatism and musicality played a tremendous role in designing these components. We created and evaluated hundreds of different implementations before deciding on the final version, which for us struck the best balance between creating a large variety of sounds and maintaining a transparent relationship between vocal input and synthesised response.

The synthesis algorithm, implemented entirely using the JSyn API (Burk 1998), is a hybrid of several techniques. The synthesizer is composed of three separate sections: an excitation source, a resonator, and a filter bank. The initial excitation is composed of two sources – a pulse oscillator and a frequency-modulated sine oscillator. These are mixed and sent through an extended comb filter, with an averaging lowpass filter and probabilistic signal inverter included in the feedback loop. The result is sent through a bank of bandpass filters and mixed with the unfiltered sound to generate the final output.

The mapper manages an entire ensemble of synthesis instruments, each of which is controlled by the vocal gestures of a single player. Much of the low-level analysis data is mapped onto the synthesizer in straightforward ways. The fundamental frequency envelope controls the frequency of the excitation sources and the length of the feedback delay line. The amplitude envelope controls multiple synthesis parameters – the amplitude of the excitation source, the overall amplitude of the synthesizer, and the depth of frequency modulation – in order to make the amplitude envelope more salient in the synthesised sound. The first and second formant envelopes are used to set the centre frequencies of the bandpass filters, and the Q on those filters are inversely proportional to the formant bandwidth envelopes. The voicedness/unvoicedness envelope modulates the probabilistic signal inverter between noisier and purer timbres.

High-level feature data, on the other hand, is used to control timbral aspects of the synthesizer which remain constant for the duration of each gesture: the ratio of pulse to sine generators in the excitation source, the probability of inverting the feedback signal, and the filter Q values. In the latter two cases, a

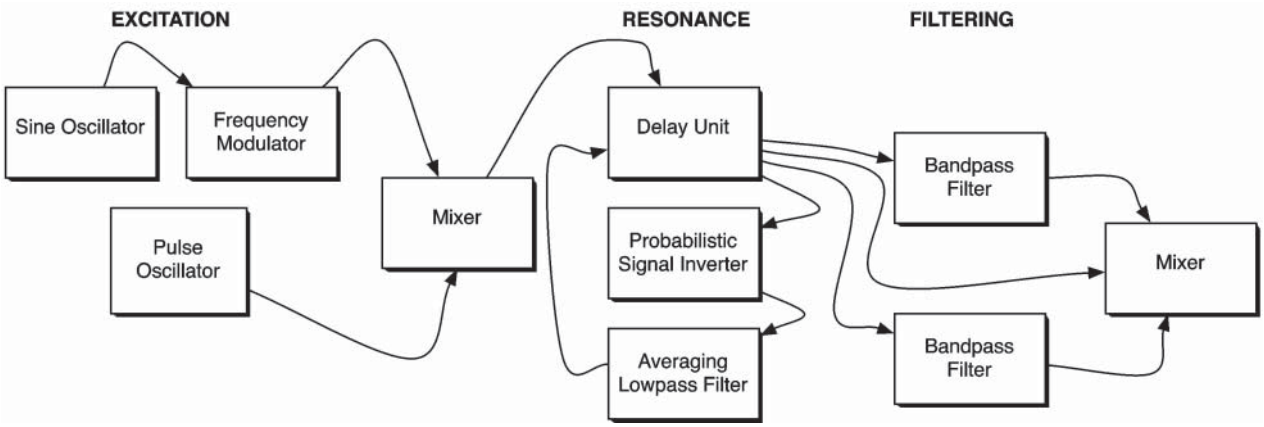

Figure 3. Synthesizer schematic.

high-level feature value defines a range within which the parameter can vary over the course of the gesture; then low-level envelopes control continuous, subtle variations within that range.

We did not want sound output to stop completely when users were not making vocal gestures. So once a player's synthesizer is finished with a gesture, it sounds a quiet 'after-ring' until the user's next gesture begins. The relationship of the vocal gesture to this after-ring is less transparent than with the gesture itself; it is a quiet sound, constantly but subtly changing, based on the slowed-down formant envelopes of the previous gesture played out of phase with each other.

To help users more easily distinguish among the sounds controlled by different players, we make small modifications to timbre and panning for each synthesis instrument. A player's own synthesizer is always panned to the centre of the stereo mix; other players are panned to the left or right to varying degrees. And the frequency modulation ratio for each player's synthesizer is randomly initialised to a different position in a lookup table, which defines an ordered set of ratios moving from whole numbers to increasingly complex fractions. Over the course of a session, the frequency modulation ratio does change, but its initial value is always unique.

3.6. Graphical user interface

The focus of Auracle is on aural interaction, so the software's graphical user interface is deliberately utilitarian and sparse. The main display area shows information about all users in the active ensemble of players: their usernames, their approximate locations on a world map (computed with an IP-to-location service), and a running view of the gestures they make (displayed as a series of coloured squiggles corresponding to amplitude, fundamental frequency, and formants). Users push and hold a large play button when they make a vocal gesture. Additional controls allow them to move among ensembles, create new ensembles, and monitor and adjust audio levels. A text chat among players within the ensemble is available in a separate pop-up window.

4. DISTRIBUTED DEVELOPMENT PROCESSES

Not only is Auracle itself a collaborative, networked instrument, but it was developed through a collaborative, networked process. The six-member project team had members based in Germany, Italy, California and Arizona. By creating a flexible, component-based system architecture and by using off-the-shelf tools and custom solutions to collaboratively develop and evaluate those components, we were able to effectively work together throughout the year-long development process to make the project a reality.

4.1. Component-based architecture

We designed Auracle as a component-based architecture because we wanted to experiment with a variety of approaches, particularly with regards to mapping and synthesis techniques. We also needed to easily integrate source code developed by different team members.

In designing the initial system architecture, we defined component types (low-level analysis, mid-level analysis, high-level analysis, network communication, mapping and synthesis) and the data objects which flow between them, without yet deciding how each component would be implemented. The components use Java interfaces, reflection, and the observer pattern, combined with an avoidance of direct cross references, to enable specific component implementations to be mixed and matched to form a complete system. During start-up, Auracle reads a text file specifying the particular components to be used and instantiates the corresponding configuration.

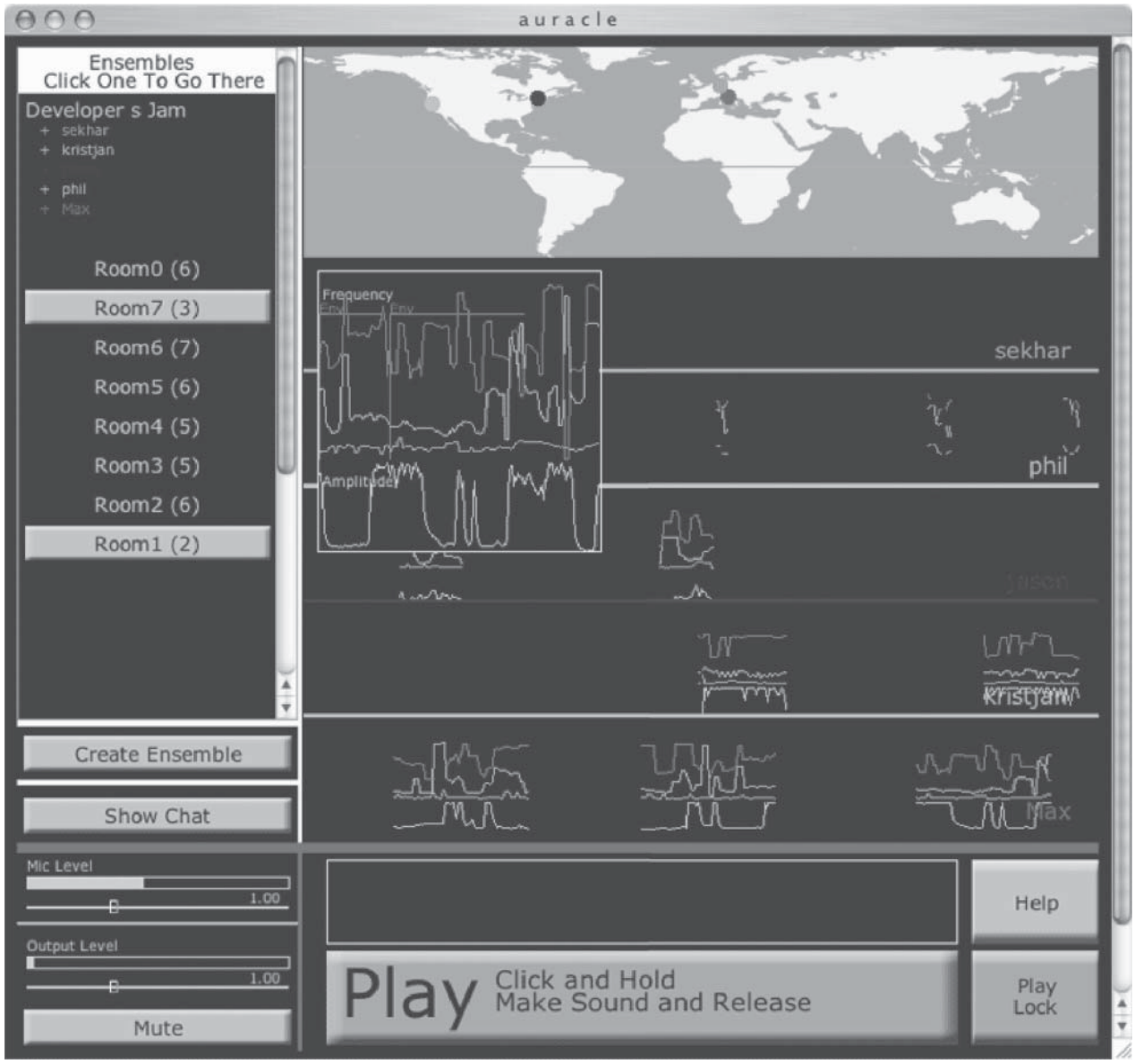

Figure 4. Auracle graphical user interface.

4.2. Auracle TestBed

While this component-based architecture helped us easily integrate new code into the system and try different implementations, it was tedious to constantly edit large configuration files and relaunch the program. As the number of experimental components grew, tracking and comparing configurations became increasingly difficult, and manually distributing configurations to colleagues became tedious.

To address these limitations, we created the Auracle TestBed, a separate application used only in the development process and not included in the public release (Varnik, Freeman, Ramakrishnan, Burk, Birchfield and Neuhaus 2004). Pop-up menus in the TestBed's GUI select analysis, mapping, synthesis, and effects unit components, and sliders adjust internal synthesizer parameters for fine-tuning control.

The TestBed saves patches, which describe component configurations and include developer annotations. The patches are saved as text files and also displayed as buttons in the graphical user interface. A single button press switches to a different system configuration, enabling rapid comparisons between patches. The change in Auracle's configuration is immediate; no text files need to be edited and the application does not need to be restarted.

From within the TestBed, developers can also upload their patches to the group development server to share them with other team members, who can use them in a group 'jam session' or download them to their local machine.

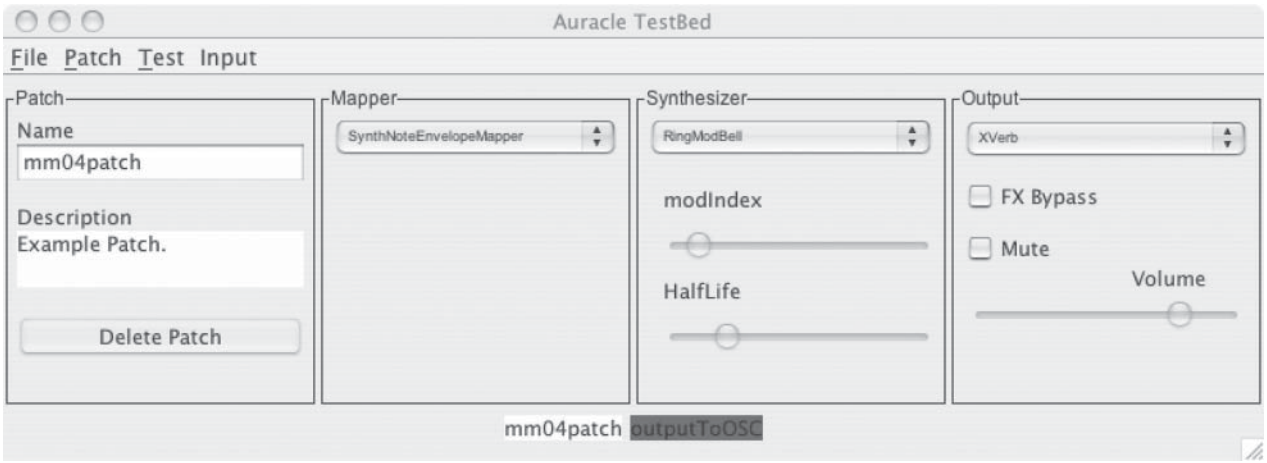

Figure 5. Auracle TestBed graphical user interface.

4.3. Rapid mapper and synthesizer prototyping

Of all the component types in Auracle's architecture, mappers and synthesizers were the subject of the most experimentation and debate; we had to implement and compare a large variety of approaches before even deciding on a general design strategy. In order to prototype these alternatives as quickly as possible, we integrated Auracle with external sound development tools.

The Auracle TestBed can send analysis data to any application which supports the Open Sound Control (OSC) protocol (Wright and Freed 1997). We used this feature to send real-time Auracle data to SuperCollider (McCartney 1996), Max/MSP (Cycling '74 2004), and Wire (Burk 2004). Each developer was able to work within a familiar, user-friendly environment which permitted runtime modifications to synthesis algorithms.

We exported patches developed in Wire as Java source code and directly integrated them into Auracle's Java source tree. For algorithms designed in the other applications, we manually ported the most successful components to Java, which was straightforward.

4.4. Group component comparison and evaluation

Auracle is designed for use by an ensemble of participants, so it was important to evaluate component implementations in group situations. Mapping and synthesis components sounded dramatically different when used individually than when used in a group 'jam session'.

So Auracle itself became a primary platform for our own collaboration on the project. Within just a few weeks of beginning development, we completed bare-bones implementations of each component type, along with text-based chat functionality, and began holding twice-weekly 'jam sessions' on our development builds. These jams, which were usually followed by Internet-based audio conference calls, were critical opportunities to evaluate our progress and discuss important issues. They also helped us to regularly experience Auracle as users rather than as developers.

Sometimes, an individual developer needed still more time to evaluate component implementations in group situations, so we developed a Headless Client to simulate user activity. In order to reduce CPU usage, the Headless Client pre-analyses audio files and stores data in a form ready to transmit to the server. It references this pre-processed data when 'jamming' on Auracle. And it does not perform any mapping or synthesis on data received back from the server.

A command-line application launches several Headless Clients simultaneously to simulate one or more ensembles of participants. A developer can simulate dozens of users from a single machine and then launch a single instance of the complete applet to 'jam' with them interactively.

5. EXTREME PROGRAMMING PRACTICES

Networked software development necessitated not only good collaboration tools, but also good development habits to keep code clear, integrated and synchronised. We followed many of the practices encouraged by the Extreme Programming (XP) paradigm (Beck 1999), including nightly automated builds and unit testing on our development server, and frequent developer collaboration and task rollover. We also used Javadoc functionality to create self-documenting source code and complemented this documentation with higher-level design overviews on our group Wiki.

Ultimately, we found that the biggest challenges in developing a robust networked music application were

to reproduce and debug problems which arose from specific vocal gestures or group activities, and to identify problems which remote users encountered but were unlikely to report to us.

5.1. Automating user input

In order to consistently reproduce problems and to create effective unit tests, we had to simulate user vocal input from pre-recorded audio files. Our TestBed application enabled us to manually select and loop through audio files which replace microphone input into Auracle, and we maintained a large vocal gesture database to use in this regard. A second, smaller collection of sound files documented gestures which caused problems such as inaccurate analyses, overloaded synthesis filters, or even crashes. We used these files to consistently reproduce problems as we were trying to fix them. Unit tests relied on the same code base as the TestBed to simulate user input.

With our Headless Client (see section 4.4 above), we were able to simulate many simultaneous users with a single computer. This helped us track bugs which only occurred in group situations and to benchmark the server under heavy usage loads.

5.2. Tracking mechanisms

In a live concert of interactive computer music, the fear is that the software will suddenly fail, the machine will crash in the middle of the performance, and there will be no time to fix it. With an ongoing Internet work such as Auracle, there is always an opportunity to fix problems; the fear is that issues will never be reported by users, and developers will never know they exist. To avoid this situation, we automatically log detailed information about problems as they occur, so that we are not dependent on users to report issues themselves.

Web server logs provide basic information about site visitors, and the TransJam server tracks some rudimentary information about user sessions. But these tools do not provide an adequate level of detail. So the Auracle applet complements this data by uploading additional information to a server-side database, tracking each client's operating system, web browser, Java implementation, and all client-side error messages and Java stack traces. The database is searchable via a Web interface, and daily e-mail summaries are sent to our mailing list.

This logging data helps us more easily track and fix bugs. When users do send us problem reports, we can quickly locate their session in the database to find information about their system configuration and error messages. And we can look directly in the database to find errors which were never reported. Often, a stack trace points us to a specific line of source code and an easy solution.

6. DISCUSSION

Auracle was officially launched to the public in October 2004 – on the Internet, at Donnaueschinger Musiktage in Germany, and during a live radio event on SWR. We have been thrilled to see how Auracle engages people ranging from non-musicians to trained singers, of many different ages and cultural backgrounds. Inevitably, some players are shy, have difficulty thinking of vocal gestures, and quit after a few minutes. But we have observed many players interacting with Auracle for over thirty minutes, enjoying the identification of their voices in the sounds Auracle produces and the surprise in hearing those transformations and the responses of others. Over time, Auracle encourages them to create an increasing variety of vocal sounds – whether whistling, gurgling, shouting or singing – as they strive to explore the boundaries of the system.

In our own regular 'developer jams' on the system, we developed strategies for collaboratively structuring our interaction over extended periods of time. Often, one of us would suddenly start making gestures which radically departed from the current sounds in frequency, density, noise content, or dynamics, and the rest of the players would gradually begin to imitate them. And sometimes, we would use Auracle's text chat functionality to plan such changes more carefully.

6.1. User base

We are encouraged by the fact that Auracle users are engaged with the system for extended periods of time, and that many of them return to participate again. During the eight months beginning 15 October 2004 there were 1,494 user sessions on Auracle, with 803 usernames connecting from 717 distinct hosts. The average session length was 14.8 minutes.

The majority of users play Auracle alone. Auracle is engaging when played in this manner, but it is most interesting when users are online at the same time and can 'jam' together. We have created perpetual, virtual ensembles on Auracle where users can play with robots if they wish, but this is not a substitute for interaction with other human beings. We have also experimented with a variety of strategies to help users find each other online, including scheduling regular online events and encouraging users to plan Auracle meetings with friends, but these techniques have had limited success. Our most successful Auracle events, ironically, have taken place in the physical world, with several computers set up as kiosks on which people can try it. We are continuing to present Auracle in this format, and we are also exploring the possibility of permanent kiosks in museums and other public spaces.

In the long term, we hope to draw enough users to Auracle so that multiple players are always online. In

this regard, we are focusing not only on drawing more users to the site, but also on getting more of them to log in and participate once they arrive. We designed Auracle with easy set-up in mind, and we tested extensively for compatibility on a wide variety of platforms and configurations. But during the eight-month period beginning 15 October 2004, 7,877 distinct hosts visited the Auracle website, yet only 717 of those hosts – just under ten per cent – actually launched and logged in to Auracle. And over half of the users who did log in never actually input a sound. All told, only about five per cent of people who visited the web site actually used Auracle!

Some users are likely perplexed by the user interface, and we are working to improve site documentation and help them more easily test and configure their audio system. Others are too shy to contribute, preferring to lurk. But our informal polling indicates that the majority of users simply lack computer microphones. While we do not expect users to buy an external microphone or headset just to use Auracle, we are encouraged by the growing popularity of online audio chat and telephony applications, and we hope that computer microphones will soon be ubiquitous even on desktop machines.

Many users are also reluctant to install the JSyn plugin, which requires them to click through a few Web pages and dialogue boxes, restart their web browser, and navigate back to the Auracle site. Though the process only takes a minute, it is still more of a nuisance than most casual Web surfers are willing to endure. Unfortunately, JSyn's reliance on native code makes automatic installation impossible, and there is currently no viable alternative which runs inside a Web browser. As computer processors continue to get faster, it will eventually become possible to implement projects like Auracle in pure Java, making such installation procedures unnecessary. Until then, we must make the best of the situation as it stands.

6.2. Opening Auracle to the computer music community

Auracle was designed for a lay public without formal musical or technical training, and those users have been the focus of our efforts to date. Now, we want to make the project more accessible to members of the computer music community. We are preparing much of the Java source code for release under an open-source licence, so that others may incorporate our development work in their own projects.

We are also interested in allowing third-party developers to contribute custom mapper and synthesizer components which would function as plug-ins to Auracle. We are creating a Software Developer's Kit to enable experienced Java and JSyn developers to participate. It will include a streamlined Java API for Auracle plug-ins, through which player analysis data can be obtained from the system and an audio stream can be returned. A stand-alone Auracle test environment will also be available.

By opening Auracle development to new contributors, we hope that the project will evolve in new ways and new directions which we could not have envisioned ourselves.

REFERENCES

Antares Audio Technologies. 2004. Antares kantos. http://www.antarestech.com/products/kantos.html

Banse, R., and Scherer, K. 1996. Acoustic profiles in vocal emotion expression. *Journal of Personality and Social Psychology* **70**(3): 614–36.

Barbosa, A. 2003. Displaced Soundscapes: a survey of networked systems for music and sonic art creation. *Leonardo Music Journal* **13:** 53–9.

Barbosa, A., and Kaltenbrunner, M. 2002. Public Sound Objects: a shared musical space on the Web. *Proc. of the Int. Conf. on Web Delivering of Music 2002*, pp. 9–15. Darmstadt, Germany: IEEE Computer Society Press.

Beck, K. 1999. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley.

Böhlen, M., and Rinker, J. 2004. When Code is Context: experiments with a whistling machine. *Proc. of the 12th ACM Int. Conf. on Multimedia*, pp. 983–4. New York: ACM.

Brown, C. 2003. Eternal Network Music. http://crossfade.walkerart.org/brownbischoff2/

Brown, C., and Bischoff, J. 2003. Indigenous to the Net: early network music bands in the San Francisco Bay area. http://crossfade.walkerart.org/brownbischoff/

Bryan-Kinns, N., and Healey, P. 2004. DaisyPhone: support for remote music collaboration. *Proc. of the 2004 Conf. on New Interfaces for Musical Expression*, pp. 29–30. Hamamatsu, Japan: ACM.

Burk, P. 1998. JSyn – A real-time synthesis API for Java. *Proc. of the 1998 Int. Computer Music Conf.*, pp. 252–5. Ann Arbor, MI: ICMA.

Burk, P. 1999. WebDrum. http://www.transjam.com/webdrum/

Burk, P. 2000. Jammin' on the web – a new client/server architecture for multi-user performance. *Proc. of the 2000 Int. Computer Music Conf.*, pp. 117–20. Berlin, Germany: ICMA.

Burk, P. 2004. Wire: a graphical editor for JSyn. http://www.softsynth.com/wire/

Cowie, R., Douglas-Cowie, E., Tsapatsoulis, N., Votsis, G., Kollias, S., Fellenz, W., and Taylor, J. 2001. Emotion recognition in human-computer interaction. *IEEE Signal Processing Magazine*, January, pp. 32–80.

Cycling '74. 2004. Max/MSP. http://www.cycling74.com/products/maxmsp.html

Diamantaras, K., and Kung, S. Y. 1996. *Principal Component Neural Networks*. New York: John Wiley and Sons, Inc.

Duckworth, W. 2000. The Cathedral Project. http://cathedral.monroestreet.com/

Dudley, H. 1939. Remaking speech. *Journal of the Acoustical Society of America* **11**(2): 167–77.

Freeman, J., Ramakrishnan, C., Varnik, K., Neuhaus, M., Burk, P., and Birchfield, D. 2004. Adaptive high-level classification of vocal gestures within a networked sound environment. *Proc. of the 2004 Int. Computer Music Conf.*, pp. 668–71. Miami, FL: ICMA.

Grey, J. 1977. Multidimensional perceptual scaling of musical timbres. *Journal of the Acoustical Society of America* **61**(5): 1,270–7.

Joyce, D. 2005. Get your own show. http://www.negativland.com/nmol/ote/text/getoshow.html

Kung, S., Diamantaras, K., and Taur, J. 1994. Adaptive Principal Component EXtraction (APEX) and applications. *IEEE Transactions on Signal Processing* **42**(5): 1,202–17.

Machover, T. 1996. *The Brain Opera*. http://brainop.media.mit.edu

McCartney, J. 1996. Supercollider: A new real-time synthesis language. *Proc. of the 1996 Int. Computer Music Conf.*, pp. 257–8. Hong Kong: ICMA.

Neuhaus, M. 1990. Audium, Projekt für eine Welt als Hör-Raum. In E. Decker and P. Weibel (eds.) *Vom Verschwinden der Ferne: Telekommunikation und Kunst*. Cologne: Du Mont.

Neuhaus, M. 1994. The Broadcast Works and Audium. In *Zeitgleich*. Vienna: Triton. http://auracle.org/docs/Neuhaus_Networks.pdf

Oja, E. 1982 A simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology* **15:** 267–73.

Oliver, W. 1997. *The Singing Tree, A Novel Interactive Musical Interface*. M.S. thesis, EECS Department, Massachusetts Institute of Technology.

Rabiner, L., and Schafer, R. 1978. *Digital Processing of Speech Signals*. Englewood Cliffs, NJ: Prentice-Hall.

Radio Show Calling Tips. 2005. http://www.pressthebutton.com/calling.htm

Ramakrishnan, C., Freeman, J., Varnik, K., Birchfield, D., Burk, P., and Neuhaus, M. 2004. The Architecture of Auracle: a real-time, distributed, collaborative instrument. *Proc. of the 2004 Conf. on New Interfaces for Musical Expression*, pp. 100–3. Hamamatsu: ACM.

Roads, C. 1996. *The Computer Music Tutorial*. Cambridge, MA: MIT Press.

Rubner, J., and Tavan, P. 1989. A self-organizing network for principal-components analysis. *Europhyics Letters* **10**(7): 693–8.

Sanger, T. 1989. An optimality principle for unsupervised learning. In D. Touretzky (ed.) *Advances in Neural Information Processing Systems*. San Mateo, CA: Morgan Kaufman.

Tanaka, A. 2000. MP3Q. http://fals.ch/Dx/atau/mp3q/

The User. 2000. Silophone. http://www.silophone.net

Varnik, K., Freeman, J., Ramakrishnan, C., Burk, P., Birchfield, D., and Neuhaus, M. 2004. Tools Used While Developing Auracle: a voice-controlled, networked instrument. *Proc. of the 12th ACM Int. Conf. on Multimedia*, pp. 528–31. New York: ACM.

Wright, M., and Freed, A. 1997. Open sound control: a new protocol for communicating with sound synthesizers. *Proc. of the Int. Computer Music Conf*, pp. 101–4. Thessaloniki, Hellas: ICMA.

Yacoub, S., Simske, S., Lin, X., and Burns, J. 2003. Recognition of emotions in interactive voice response systems. http://www.hpl.hp.com/techreports/2003/HPL-2003-136.html