Jammin' on the Web - a new Client/Server Architecture for Multi-User Musical Performance

Philip L Burk, philburk@softsynth.com
http://www.transjam.com

ABSTRACT

This paper describes a software system called the **TransJam Server** that allows several musicians to log into an Internet web site and perform music together. The system is general purpose so programmers can develop web based applications in a variety of styles. A single server can support several different applications simultaneously, for example musical applications, games, or conferencing. A simple protocol allows clients to enter a lobby, obtain a list of active sessions, join a session, and then exchange information with other members of the session. Server operations such as object locking and data broadcasting allow users to create and edit shared data objects without corrupting them. The server is written in 'C' and runs on PC or Unix based hosts with low overhead.

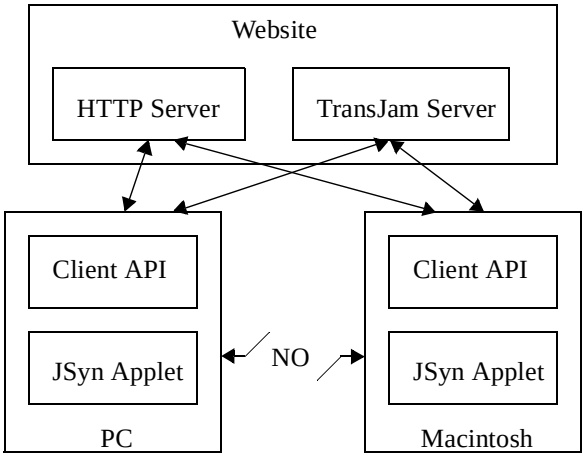
Client applications are typically written in 'C' or Java. A chat feature is available which allows musicians to discuss musical strategies and to interact socially. The Java version can be used with a number of audio APIs including JSyn, JMIDI, or SUN's new JavaAudio. Applications must be designed to work well despite Internet latencies. Examples include drum boxes, algorithmic music generators, or “ambient” synthesis environments.

1 INTRODUCTION

The goal of this project is to enable musicians to perform together through the Internet. Surfing the web is usually a very lonely activity. There may be hundreds of people visiting a web site simultaneously, but there is often no way for them to communicate. Chat software allows people to exchange text messages in real-time. And game sites provide activities besides chatting, like chess or card games. But performing music together on the web is difficult because of two major obstacles. One is the lack of high fidelity sound generation in the default browser. The JSyn[1] Audio Synthesis API was designed to address that need. The other obstacle is that Java Applets running in a web browser need a way to find each other, and exchange musical information. TransJam was designed to address those needs.

Other systems have recently been developed for network based music interaction. Yamagishi & Setoh [4] developed a system that controlled Max running on a server through CGI scripts submitted from a browser. The resulting music was broadcast back to the user via RealAudio.

This project was inspired by the work of the pioneering network band, “The Hub”.[2] The Hub consisted of Chris Brown, John Bischoff, Scot Gresham-Lancaster, Tim Perkis, Phil Stone and Mark Trayle. They performed live interactive compositions that involved exchanging information through a star network with a central hub. Members would propose a network protocol for a piece that would then be implemented by each member on their individual computers. My hope is that this server will enable composers to create networked compositions in the style of the Hub and to run them on the web.



2 COMPONENTS

2.1 Server Software. The TransJam Server must run on the same computer that serves the web pages. This is because Java Applets can only open a socket on that host computer. Applets are prevented from connecting to a socket on other computers by the security mechanisms in Java and the browser.

One of my design goals was to make the server as simple as possible. The server, therefore, only provides functions that can be implemented by a server but not by a Java Applet. These functions are similar to the functions that are provided to processes by a multi-tasking operating system. They include semaphores, inter-client communication, synchronization and record locking.

The TransJam Server is written in ‘C’ and uses the BSD socket library.[3] It was originally coded as a multi-threaded application with one thread per client. But I was able to reimplement it as a single thread by using the select() function to wait on multiple clients. This made the code more robust and simplified the coding of atomic operations such as record locking.

The server waits for new clients to log in, or for existing clients to send messages. When a message is received it is processed and the server may then send messages to one or more clients. The socket connection is maintained the entire time the client is logged in so that the server may send any client messages at any time.

A client is always associated with a *session*. A session contains a list of clients and a list of *things* that can be shared among the clients. When the client first logs in, they are placed in a top level session called a *lobby.* A single server can support multiple applications, like chess or a webdrum. Each application has its own unique lobby.

Each lobby can have multiple child sessions. Users can either create their own new session, or join an existing session. Sessions can be nested arbitrarily deep under the control of the application.

Clients can communicate with other clients in the same session by sending messages through the server. A simple example would be a chat application that sent text messages to all of the users in the session.

2.2 Shared Data Base in RAM. If clients wish to edit a shared data structure, for example a melody or drum pattern, then they must ensure that their edits do not interfere with each other. If the edits are not properly synchronized, then one persons changes may be lost. The same problems can occur in a company data-base that is being updated by multiple persons. The server, therefore, maintains what is essentially an in-memory data base. Each record, or thing, has a name and an associated string that contains the state of the record. In order for things to be edited, they must be owned by a client. The server manages ownership and prevents clients from editing things that they don't own.

When clients leave a session, the things that they owned are marked as not owned but are not deleted because they may still be in use by the application. When the last client leaves a session, however, every thing is deleted and the session is then deleted as well. The server does not keep data that is not being used, and writes nothing to disk except trace logs.

2.3 Java Client API. A set of Java classes is provided that handles communication with the server. A separate thread waits for messages from the server, parses the network packet, and then calls back to a *listener* with the message. A subclass of Panel is provided that manages the GUI and the transactions involved in logging in to the server and selecting a session.

2.4 Java Applications. The programmer can use the server from a Java application, or an Applet that runs in a browser. No plug-ins are required to use the server because network transactions with the hosting server are allowed by the Java security manager.

3 CLIENT/SERVER PROTOCOL

3.1 Message Format. All messages sent between the client and server have the same format.

Key - byte that identifies start of message.
Command - byte opcode.
PayloadSize - short containing size in bytes of optional text string.
Num1,2,3,4 - longs that hold message parameters such as session IDs, client IDs, etc.
Payload - optional byte array up to 2**16 in length for names and encoded state information.

3.2 Commands between Client and Server. There are approximately 50 different commands that can be sent between the client and server. Here is a partial list of the most important commands:

- create/join session,
- request list of available sessions,
- create thing,
- lock/unlock thing for editing,
- modify/query thing,
- send specific user a private message,
- send message to all users in same session (chat),
- error reports from server such as "command out of context" or "invalid ID",
- administration commands such as querying number of clients and sessions, memory usage, shutdown, etc.

4 DESIGN ISSUES

4.1 Latency and Bandwidth. Latency is defined, in this case, as the maximum time it takes to send a small packet from the client to the server, and to receive a reply. The server is pretty responsive as long as it is not swamped with users. So the bulk of the latency is due to the round-trip travel over the Internet. Here are some Internet latencies measured using PING from my office near San Francisco**:**

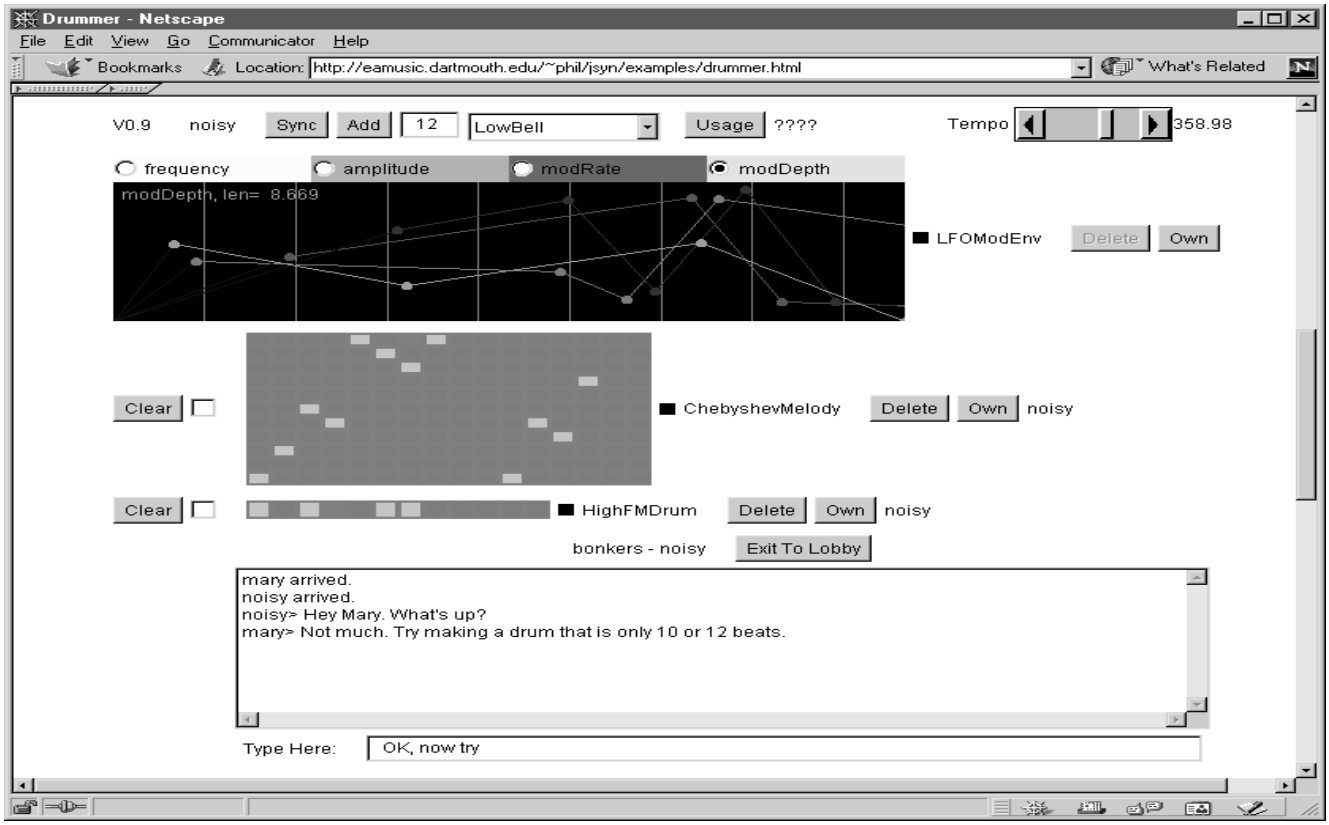
Table 1: Typical Latency

Location	Milliseconds
my ISP, California	13-140
CNMAT, California	90-300
ATR, Japan	220-350
STEIM, Netherlands	220-400
QUT, Australia	330-500

Because a latency of 100 to 300 milliseconds is very typical, it is not practical to engage in a traditional real-time "jam" where musicians play "in time" with each other. So the interaction between musicians must take place "out of time" where the arrival time of a message is not critical. The most suitable type of applications are ones that involve indirectly controlling a real-time music generating process. Examples include editing a looping data structure such as in a "drum box", or modifying parameters that control an algorithmic sequence generator.

Since we are only sending control information and not PCM audio data, bandwidth is not an important issue, even over a 56K modem.

4.2 Simultaneity and Synchronization. In applications where the users are editing a looping data structure from



remote locations, it is not necessary for them to be synchronized. One person may be hearing the beginning of the loop when the other is hearing the middle. But in some cases, more than one user may be in the same location. This occurred at the NUMUS 2000 festival where we had four Macintoshes playing the WebDrum in the same room. For this event I added a feature that allowed users in a session to synchronize themselves.

The synchronization is triggered when a user hits the "Sync" button in the Applet. This causes a message to be sent to the server with a command to relay the message to all of the clients in the session, including the requester. Encoded in the message is the current beat index of the user who hit the Sync button. As each client receives the message they force their timing to that specific beat. Some of the clients may be in locations that are near or far from the server, so the messages will not all arrive at the same time. But is likely that the messages will arrive at nearly the same time for computers in the same room. Thus the computers who are within earshot of each other will be synchronized reasonably well.

4.3 Impact on Host Computer. Servers are often run on systems shared by many users, and should have minimal impact on the host computer. I initially wrote the server in Java which was a very pleasant experience. But getting a Java program to run on my ISP's computer was problematic. I had to fit within a 300 KB RAM limit, but a minimum Java program consumed over 3 MB. Also some host computers do not have Java available so I decided to rewrite the server in 'C' which is quite portable for this type of application. The server consumes an imperceptible amount of CPU time to run the WebDrum but the amount would obviously increase with the number of clients.

4.4 Scalability. The current server is limited in the number of clients it can handle by the number of open sockets that can be created. It is currently a two tier architecture where all clients are handled by one server. To scale to larger applications, a three tier architecture would need to be developed where clients communicated with multiple servers on the middle tier. The mid level servers would concentrate information and pass it up to a single very large server at the top level.

5 .WEBDRUM EXAMPLE

5.1 Program Overview. The WebDrum was designed to test the capabilities of the TransJam server. It is based on a traditional drum pattern editor where the user turns on or off notes on a grid. The drum sounds are synthesized using JSyn[1], a Java audio synthesis API developed by the author. JSyn has an event buffer for time-stamping sound events so the drum patterns can be played with very accurate timing. By using synthetic drum sounds we eliminate the need to download large audio sample files. JSyn is available at:

`http://www.softsynth.com/jsyn`

Drum sounds are generating using a variety of synthesis techniques including FM, WaveShaping using Chebyshev

polynomials, filtered sawtooth waves, ring modulation for bells, and a Karplus-Strong plucked string sound.

5.2 User Interface. Users can add instruments of various types:

- Drums = single pitch, turn on/off beat,
- Melody grids = N pitches in a pentatonic just intonation, 1:1, 5:4, 4.:3, 3:2, 5:3
- Envelopes = editing contours of a parameter, for example, frequency.

The program uses a Java *interface* for instruments that is very generic. This will make it easy to add algorithmic pattern generators and other types of "drums" to the mix.

The number of beats in the pattern can be selected before creating the instruments. This allows musicians to experiment with poly-rhythms. At the moment, all notes have the same length.

The "Own" button allows a user to grab an instrument from another player. This uses the record locking feature of the server which prevents editing collisions. The name of the owner of a drum is displayed beside the drum.

I placed a check box to the left of the drum that reverses the direction of the playback. I intentionally left it unlabeled so that people could discover its function.

6 LESSONS LEARNED

6.1 NUMUS 2000 Festival. I observed many people using the WebDrum at an installation for the NUMUS 2000 Festival in Aarhus. The program was quite stable and was well received by the festival goers. Many people were jamming locally but there were few US participants because of the time zone difference. Most users exercised all of the features and seemed to enjoy the social aspects of the chat feature while they were jamming.

As usual, no one read the instructions so most of the user training came from people standing around, or through the chat window. This installation at NUMUS revealed a number of areas where the program could be improved.

6.2 The Two-Step Problem. There are some parts of the GUI that require people to perform two steps in order to get one result. One is adding a drum. They must currently select their drum from the list of drums. Then hit the "Add" button to add the drum to the list of available drums. This completely stumped many users. So I may start with a pre-loaded set of drums. When users get proficient they could then add or subtract drums using a one-step process. Joining existing sessions also requires a "select then join" process which could be replaced by a simple set of buttons that will join a session in one click.

6.3 Screen Area. I am not making very efficient use of the screen. As people add drums, particularly melody grids and envelopes, the chat box gets pushed off the screen. This makes it hard sometimes to get a message to a person when desired. I need to conserve screen space and may move the chat box to a separate window that floats over the drums.

6.4 No One to Play With. The WebDrum is not yet a popular web site. So when people log in they often have no one to play with. This discourages them from coming back. This is a positive feedback cycle that must be broken in order for a multi-user site to succeed. I plan to add an appointment system so people can schedule a time to come back and meet with others. Scheduling would have to be based on Greenwich Mean Time to accommodate different time zones. Another technique might be to schedule special events when several people are likely to show up at the same time.

I have a DSL line to my office so I can sometimes leave a machine playing an empty drum pattern. If someone logs in and starts making sound I can hear them and join in.

6.5 Idle Lines get Dropped. Some phone connections have the annoying property of disconnecting if there is no activity for a certain length of time. This can happen if people stop playing and just listen to a piece for a while. Since the server expects the client to always be connected, the client is logged out permanently when the line is dropped. I may either add a feature that allows a client to log back in and recover their session if disconnected, or provide an option to ping the server periodically to keep the line alive.

7 AVAILABILITY

The server software runs on Windows and UNIX systems. The client runs on any system that supports the standard Java platform. Binary versions for a limited number of clients can be obtained for free from the author for non-commercial research and artistic purposes.

REFERENCE

[1] Burk, Phil. 1998. "JSyn - A Real-time Synthesis API for Java" *Proceeding of the International Computer Music Conference 98*:252-255

[2] Hub Network Band, collected scores at "`http://tesla.csuhayward.edu/history/08_Computer/Hub/hubspecs.html`", CDs at "`http://www.artifact.com/wreckball.html`"

[3] Stevens, W. Richard, *UNIX Network Programming, Volume 1: Networking APIs - Sockets and XTI*, 1997, Prentice Hall; ISBN: 013490012X

[4] Yamagishi & Setoh, 1998 "Variations for the WWW - Network Music by Max and the WWW", *Proceeding of the International Computer Music Conference 98*:510-513