CPS 512.01 Spring 2015 Final Project
Synchr

Bobo Bose-Kolanu, Miles Oldenburg
Duke University

April 22, 2015

Contents

1 Introduction and Motivation

Synchr is a distributed music sharing web application. It combines a RESTful API with intelligent use of node.js's asynchronous callback feature and the socket.io library's broadcast mechanism to enable users to share music libraries and distribute player control commands.

In this fashion users can browse each other's music libraries and control distribution ensures that when one user hits play all users hear the same track at the same time. Original work required the construction of a scanner.js routine that could recursively scan the music directory of a user's computer, scrape the tracks for id3 information, and upload them in an HTML compliant fashion for presentation within the DOM. Additional original work resulted in the production of an internal server-client API that allowed clients' tracklists to be updated on join/exit of nodes and allowed play controls (pause, skip, play) to distribute to all connected clients.

Synchr provided us with an opportunity to learn more about web application frameworks, explore augmentation of the traditional client-server model with distribution, apply lessons learned about shared state and distributed systems from class, and gain experience in producing client-ready implementations. The rest of this document outlines the Technology (software stack), Server Architecture, Client Architecture, Current Limitations, and directions for Future Work.

2 Individual Contributions

Before we begin the technical report on our project we pause to list the individual contributions each team member made

Abhishek Bose-Kolanu

- Added server logging of client IP addresses as they issue request track commands
- Removed hardcoded server connection and added dynamic server connection (scrapes IP and port from browser upon user input)
- Added stale track removal feature. Captures disconnecting node IP on disconnect event and signals surviving nodes to remove tracks previously served by the disconnected node

- Produced initial PowerPoint and outline for class presentation and presented in class
- Wrote Introduction and Motivation section and first draft of Server section as well as two sub-sections of Current Limitations.
- Proofing and copy-editing of final report

Jesse Hu

- Prepared slides on network time synchronization for class presentation

Miles Oldenburg

- Project setup including structure, dependencies, documentation, and installation scripts
- Implemented both the client and server nodes including the interface, distributed control, track scanning, track streaming, and node communication
- Proofing, code examples, and topology slide for class presentation
- Final report including the setup and Server, Client, Current Limitations, and Future Work sections

3 Server Nodes

3.1 Overview

A server network can be comprised of one or many nodes. Each server node may have many or no clients attached. Server nodes serve several functions.

Serve Client Interface
Nodes listen for client connections and serve as web and application servers delivering the client interface as static HTML and JavaScript.

Music Discovery
Server nodes perform a recursive scan of a designated directory on their disk to discover music and tag it for readability in the client interface.

Tracklist Sharing
When a new server node joins an existing network, it sends its tracklist of the tracks it has scanned to the network. The network then broadcasts this information to all other server nodes and in turn to any connected clients where it is automatically added to their interface.

Tracklist Removal
Since only server nodes share tracklists, removal of stale tracks occurs when one server node quits the network. On disconnect its IP address will be captured and sent via a socket broadcast to connected clients. These clients will receive the stale-track event and remove the tracks from their interface whose CSS id's match the IP of the disconnected server.

Streaming
Server nodes enable on-demand streaming of music tracks. Client interfaces request music and the server node streams the file from disk over HTTP.

Player Controls
Server nodes handle the distribution of control commands through broadcast. Control commands like play, pause, and skip are received and then broadcast through the network reaching all other connected clients.

3.2 Technology

The server nodes use node.js[1], an event driven, non-blocking JavaScript framework that is useful for scalable network applications. Each server runs express[2] for the web and application servers, and socket.io[3] for web socket communication. A full detailed list of node.js modules can be found in the package.json file in the repository.

[1]https://nodejs.org/
[2]http://expressjs.com/
[3]http://socket.io/

4 Client Nodes

4.1 Overview

Client nodes are the web application consumers. A client can join any of the server nodes if they know the IP address. For example, a client would go to http://192.1.0.101:3000/ in their browser and be served the interface if a server node is running on that address. When the browser renders the interface it makes a call to the server node and retrieves the tracklist from that node. When the user clicks play on any track the audio player streams that track from the proper server node.

If the server node receives any track updates it broadcasts them to the client where new tracks are automatically added to the interface. If the server emits a stale tracks event along with a failed node address then those tracks are automatically removed from the client interface.

When the user controls the audio player with a pause, play, forward, backward, or skip that control event is emitted to the server node who then broadcasts it to the network. Similarly, the client may receive a control event from the server node and then controls the audio player in its interface accordingly.

4.2 Technology

The client interface is comprised of static assets like HTML, CSS, and JavaScript. The client was built as a web application because it can be used on a huge variety of devices without installation as long as a a modern browser is present. Bootstrap[4] is used as a UI framework and require.js[5] is used for module management. Like the server nodes, the clients run socket.io for web socket communication. Tracklist additions and removals as well as all control events are handled through socket.io. The audio player is the standard HTML5 audio player. A full list of libraries can be found in the bower.json file in the repository.

[4]http://getbootstrap.com/
[5]http://requirejs.org/

5 Current Limitations

5.1 Server Node Robustness

The server network is not completely robust to node failures. Each node can be connected to many other nodes in order to share tracklist data. The player controls are also broadcast through the server node network before continuing to all the clients. If a server node fails then any downstream servers would not receive control updates.

The solution to this problem would be for server nodes to store the successor of their successor and predecessor nodes so that the network can be reconnected in the event of a failure.

5.2 Server Node Tracklist Memory

Server nodes do not store the combined tracklist data in memory. If a new node joins the network and they receive new tracklist data, the node simply forwards the new data on to other nodes and clients. This means that if a client has joined the network after a server node it would not receive the combined tracklist for both server nodes. It would receive only the tracklist for the server node it connected to in addition to all tracklist updates from any subsequent server node joins.

The solution to this problem would be for each server node to store combined tracklist data in memory. Then when a new client connects to it, the client can immediately be aware of all tracks the server node knows of.

5.3 IPv6 Compatibility

Our use of the socket.io library in conjunction with node.js produces certain constraints regarding IPv6 and IPv4 cross-compatibility. When a client disconnects we want the servers to tell surviving clients to remove the disconnected client's tracks from their lists. We accomplish this by caching the disconnecting client's IP on disconnect and pushing it to the remaining clients.

We use the socket.io library to handle the disconnect event and scrape the client's IP address. While our server architecture operates on an IPv6 socket, in our testing environments clients (cell phones and laptops) connected using

IPv4 addresses. As a result, the socket.request.socket.remoteAddress call returns a hybrid of an IPv4 pre-pended with IPv6. For example, *::ffff:192.168.254.4*.

Based on a Linux networking discussion thread [6] , it looks like this prefix is a way for a socket to be of form IPv6 but to allow IPv4 packets to be transmitted across it. In other words, this prefix provides a way of mapping IPv4's into the IPv6 address space.

As a result, we currently detect presence of the ::ffff: and remove it if present, passing the result to a mechanism that deletes `<tr>` entries with corresponding CSS id's (since tracks are tagged in the tracklist with the IP of the client that provides them).

However, some operating systems may disallow passing IPv4 traffic over IPv6 sockets, in which case, since our application appears to be using an IPv6 socket, we will get a real IPv6 address for clients with these operating systems. Current application logic for the disconnect event handler does not discriminate between this situation and the pre-pended IPv4 above. One possible solution is to detect a true IPv6 address and simply pass it through without modification. To ensure inter-operability with our CSS id's we would have to require them to support IPv6 scraping for tagging as well, so this patch would require further testing and evaluation of the socket.io and ip libraries.

5.4 NAT Masking and Localhost Collisions

We face an additional limitation from the underlying networking libraries related to IP addressing: NAT masking and localhost collisions. If a user runs the server on their laptop and also connects as a client to the server (e.g. via localhost:3000 in the browser), then disconnect events do not capture the correct IP.

Currently the underlying network libraries provide ::1 (IPv6 for 127.0.0.1) when the localhost client disconnects. However, our scraping script, which is responsible for tagging the uploaded tracklist entries with the IP address of the client that provides them, uses the external-facing IP address. Consider the following scenario: a user runs two server nodes on their laptop, and joins them. A friend on an external machine connects as a client. If one of the server nodes exits the tracks from both server nodes will be delisted. A user might reasonably run two servers from one machine if they were attempting

[6]https://groups.google.com/forum/#!topic/comp.os.linux.networking/9uVVSOgScqw

some separation of concerns due to content restrictions (e.g. licensing). A solution might involve tagging tracks with <IP:port> pairs instead of just IP.

Additionally, our current networking logic does not handle NAT collisions. While extremely rare, it is possible for clients on an internal network to share an IP address with a client from an external network. In such a case when one of the clients disconnects both clients' tracks will be removed from remaining clients' tracklists. This problem could be addressed by tagging the tracks with IP + client ID (a hash value) instead of just IP alone.

6 Future Work

6.1 Control Opt Out

In the current version of the software, any client connected to the network will receive control messages. This feature may not be always desirable so the ability to opt out of sending and receiving control messages could be added to the client interface.

6.2 Pre-Fetch

In a normal listening session we can expect the next song in the library to play immediately after the current song has finished. In the current version, the upcoming song would only be streamed after the current song has finished. One optimization would be to look ahead in the potential play queue and fetch a number of songs into browser memory so when the current song finishes playing the next song can be played immediately.

6.3 Contributing

Source code and documentation are available at
`https://github.com/milesoldenburg/music-synchr`