DRAWSOCKET: A BROWSER BASED SYSTEM FOR NETWORKED SCORE DISPLAY

Rama Gottfried
Hochschule für Musik und Theater
Hamburg, Germany
rama.gottfried@hfmt-hamburg.de

Georg Hajdu
Hochschule für Musik und Theater
Hamburg, Germany
georg.hajdu@hfmt-hamburg.de

ABSTRACT

We present DRAWSOCKET, a new platform for generating synchronized, browser-based displays across an array of networked devices developed at the Hochschule für Musik und Theater, Hamburg. Conceived as a system for distributed notation display with applications in music and spatial performance contexts, DRAWSOCKET provides a unified interface for controlling diverse media features of web-browsers which can be utilized in many ways. By providing access to browser mouse and multitouch gesture data, and the ability to dynamically create user-defined callback methods, the DRAWSOCKET system aims to provide a flexible tool for creating graphical user interfaces. Included is a discussion of the architecture design and development process, followed by an overview of the features, and syntax considerations for the DRAWSOCKET API.

1. DRAWSOCKET

The DRAWSOCKET design approach is based on the "o.io" paradigm developed at the University of California, Berkeley's Center for New Music and Technology (CNMAT), which uses the OpenSoundControl (OSC) encoding [1] to create a uniform user application programing interface (API) by "wrapping" vendor- and protocol-specific details in an interoperable API syntax [2, 3]. In this way, the DRAWSOCKET system is an "o.io" wrapper for web browser display and interaction, aiming to provide a homogenous OSC API for manipulating the graphic building blocks of Scalable Vector Graphics (SVG), [1] Cascading Style Sheets (CSS), [2] HyperText Markup Language (HTML), [3] and a curated collection of client-based JavaScript libraries for animation and sound production.

1.1 Architecture Overview

The DRAWSOCKET architecture is structured as a server-client system, using Max [4] as the primary controller interface. From Max, control messages are sent to specified client browsers via a Node.js [5] server, which routes the messages by a given client addresses. The messages are then parsed and executed in the client-browser, to generate content or perform other actions.

The choice of Node.js for the server backend was particularly helpful due to the Node Package Manager (NPM), which is bundled with Node.js, allowing DRAWSOCKET to leverage the active community of Javascript library development of tools for browser-based display and inter-computer communication [4]. Further, NPM provides a practical method for managing libraries dependancies, via the *package.json* system.

In an effort to scale to larger groups of clients running off of the same server, a loosely defined *model-view-controller* [5] pattern is used to separate the processes. The server is used primarily to relay and cache the drawing commands, while the drawing implementation is offloaded to the client browsers, which have become quite efficient with recent developments in mobile computing. [6]. See figure 1 for an overview schematic of the system.

1.2 Controller

Max. Currently the primary targeted user server control platform is Max, which provides many algorithmic and interprocess communications tools. Since the release of Max 8, Max now includes the Node For Max (N4M) [6] framework, which embeds the Node.js server engine within the Max programming environment, accessible through a set of Max objects.

The first versions of the DRAWSOCKET system used an independent Node.js application running from the command prompt and a User Datagram Protocol (UDP) [7] socket to send and receive OSC bundles to and from Max, which were then broadcast to subscribed clients. However, after comparing benchmarks measuring the roundtrip messaging time between Max and node.js, the N4M system was found to be faster, and so we adopted this platform as the primary use case. However, the DRAWSOCKET system is well compartmentalized, and so could still easily be reconfigured for use with other control applications.

Within Max, the core Node.js server script, *drawsocket-server.js*, is run within Max's *node.script* object. For convenience, the node.script object is wrapped in Max abstraction called *hfmt.drawsocket* which aids in managing server

[1] https://www.w3.org/TR/SVG11/
[2] https://www.w3.org/Style/CSS/specs.en.html
[3] https://www.w3.org/TR/html52/
[4] https://cycling74.com

[5] https://nodejs.org
[6] https://docs.cycling74.com/nodeformax/api/
[7] https://tools.ietf.org/html/rfc768

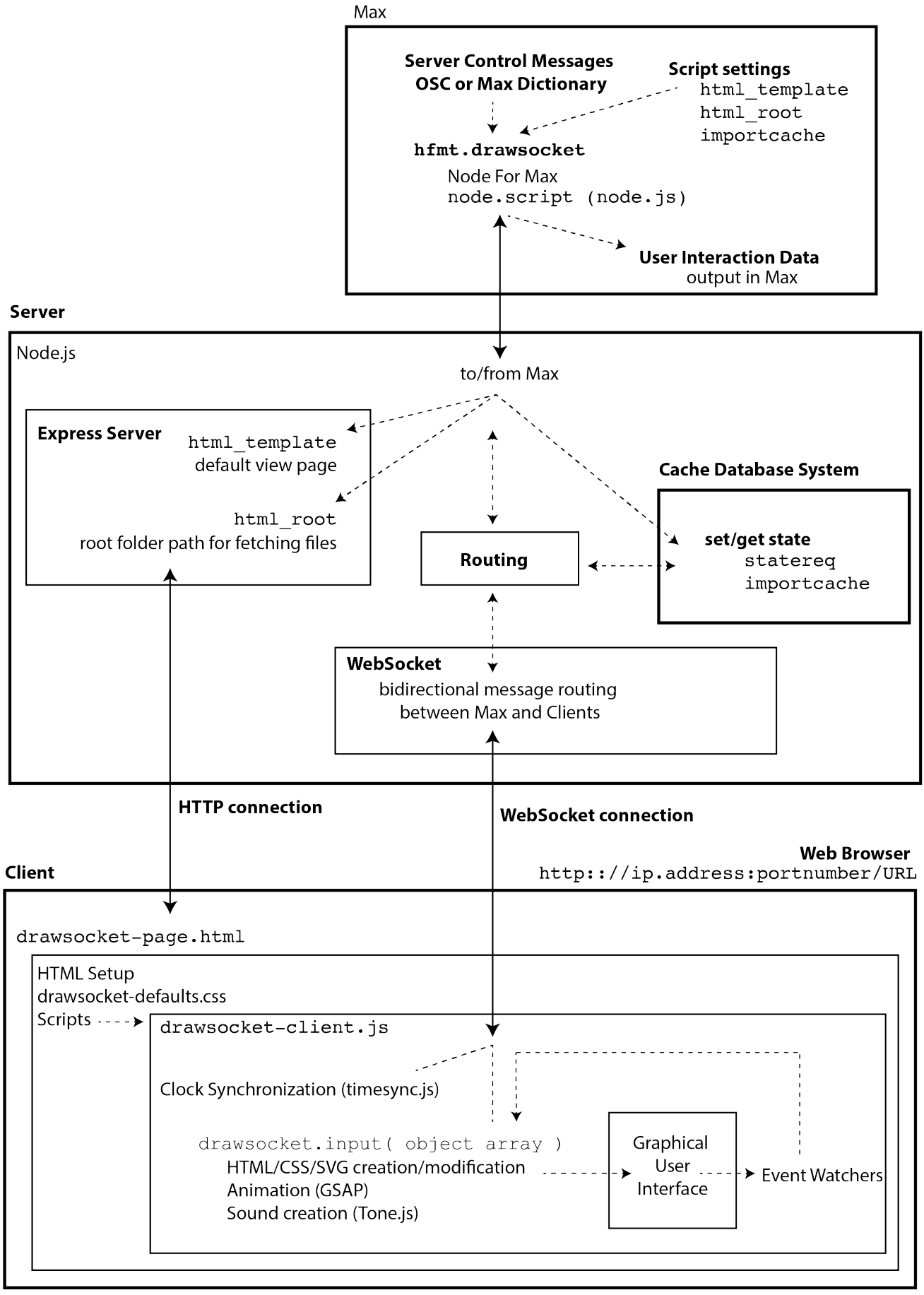

Figure 1. DRAWSOCKET Server/Client Architecture.

asset paths, and handles user interaction messages returning from the client (see section 2 for more details).

OSC-JSON representation. The DRAWSOCKET data is formatted as a key-value tree, which in Max can be represented as either OSC, or in Max Dictionary format, both of which can be easily transformed to JavaScript Object Notation (JSON). [8] See section 3 below for an in-depth description of the DRAWSOCKET messaging syntax.

1.3 Server

Node.js. The Node.js server consists of four main processes: (1) an Express HTTP server, [9] (2) a WebSocket [10] connection manager, (3) state caching, and (4) handling messages from the client (either forwarding them to the Max host environment or responding back to client, as in the case of clock synchronization and on-load initialization).

Express. The Express JS library is used to create the web server and handle HyperText Transfer Protocol (HTTP) [11] requests from client browsers. By default, the server responds to all page requests with a default HTML file which contain the basic setup necessary for most uses of the DRAWSOCKET system, with links to dependency JS libraries, fonts, and a default CSS stylesheet.

If a custom HTML page is desired, users can send the *html_template* message from the Max interface, to set a new default HTML file.

The Express server uses a static public root folder, which exposes a selected folder path that client browsers may load files from. To set the root public folder, users can set the *html_root* folder path as an initialization argument to the *hfmt.drawsocket* abstraction.

User configuration of the system is described further in section 2.

WebSockets. WebSockets are used as the primary server-client communication exchange protocol. The server accepts WebSocket requests from client browsers, and subscribes clients to receive messages addressed to their browser's URL (Uniform Resource Locator).

Control messages composed in Max as an OSC bundle or Max Dictionary are received in JSON format by the server, and routed to the clients identified by their corresponding URL web-address. [12]

URL state caching. Upon receiving control messages addressed to a new URL, the server first forwards the messages via WebSocket to the specified URL address, and then sends a copy of the message to the state cache system.

On connection to a new WebSocket, the client requests the current state of its URL from the server cache. This provides a mechanism for preloading a set of drawing commands to a given URL, so that when a user first loads the page, or hits refresh, the current drawing state of the page will be loaded.

[8] https://www.json.org/
[9] https://expressjs.com/
[10] https://www.npmjs.com/package/ws
[11] https://www.w3.org/Protocols/
[12] https://www.w3.org/Addressing/URL/url-spec.txt

Client return messages. WebSocket connection is bidirectional, and is used to handle messages from the client: responding to clock synchronization requests, initialization requests, and forwarding user interaction information to Max.

Connection Port. On startup, the server provides its IP address and connection port to the Max environment. Clients may connect remotely via network IP address, or if on the same computer, use the localhost identifier, followed by the port number (currently 3002 by default), separated by a colon (e.g. *http://localhost:3002*).

1.4 Client

Running inside a web browser, the client-side component of the DRAWSOCKET system handles the drawing and sound generating commands, clock synchronization, and user interface event watchers.

Web browser. Currently the system targets Safari and Chrome. At the current time of writing, Firefox SVG 2 feature support is behind the above two. Other browsers may also work, but are not currently being tested.

Layout. The central browser display layout consists of one HTML node, which contains an <svg> element, and an SVG <g> group node.

```
<div id="main-div">
  <svg>
      <defs id="defs"></defs>
      <g id="main-svg"></g>
  </svg>
</div>
```

The default formatting for the basic page elements is setup in the *drawsocket-defaults.css*. When users add SVG elements, they are added to the "main-svg" group. Users may also create new SVG groups and add elements to these new groups, to control stacking order, described below in more detail.

Libraries. DRAWSOCKET currently makes use of the following client-side Javascript libraries, all available from NPM:

D3.js, [13] a library for SVG information visualization. [14]

PDF.js, [15] a PDF viewer library developed by Mozilla, providing support for PDF file reading, viewing and page turning.

Tone.js, [16] a web-audio library by Yotam Mann. Currently, DRAWSOCKET uses Tone.js to provide basic soundfile playback functionality, this may be expanded in the future.

Timesync.js, [17] a clock synchronization library, used in DRAWSOCKET to synchronize animations, and provide a mechanism for timed commands.

[13] https://d3js.org/
[14] We are now mainly using d3.js for DOM node creation and manipulation. So, eventually it is likely that we will remove the d3.js dependency to streamline the codebase. However for the time being d3's utility functions are convenient for rapid development, and appear to be performant enough.
[15] https://mozilla.github.io/pdf.js/
[16] https://tonejs.github.io
[17] https://www.npmjs.com/package/timesync

TweenMax and TimelineMax, from the GreenSock Animation Platform (GSAP), [18] a high-performance system for JS and CSS animation.

Client-side Script. The client-side script running in the browser is called *drawsocket-client.js*, which handles the command processing logic of the system and its execution in the browser.

On load, the script first requests a new WebSocket connection to the server using the browser's URL address, sent to the server via the the WebSocket URL identifier (e.g. *ws://localhost:3002/violin*). On successful WebSocket connection, the script begins the clock synchronization process which runs in the background on the client system, requesting new clock readings from the server at regular intervals [19] . Once the initial clock synchronization is completed, the script sends a state initialization request to the server, to which the server responds with a sequence of commands corresponding to the current state of the given client OSC address.

The central command processing is performed in the function *drawsocket*, which parses an array of time-tagged command objects, and executes the corresponding graphic and sound manipulations in the browser. The *drawsocket* function expects one or more objects with a *key*, *val*, and *timetag* key-value pairs:

```
{
    timetag: current time (supplied by server),
    key: command string
    val: command arguments
}
```

Generally, these objects are formatted in the Node.js server from the API command messages received via Max, however they may also be created by user scripts called from event watchers.

2. DRAWSOCKET USER SETUP

The interface for DRAWSOCKET is designed for use in Max, and is distributed as a Max Package, currently hosted on GitHub at the following url:

https://github.com/HfMT-ZM4/drawsocket

To install, users download the repository and place it in Max's Packages folder. Once installed, users can instantiate the DRAWSOCKET system by creating a hfmt.drawsocket object in a Max patch.

The dependency NPM libraries are not distributed with the package, so on first loading hfmt.drawsocket, you need to send the object the *"script npm install"* message, which asks NPM to download all of the dependencies listed in the node project's *packages.json* file. [20]

Setting the public folder. As mentioned above, by default the DRAWSOCKET server responds to HTTP URL page requests with a default HTML page. Custom HTML pages, and/or other types of assets can also be served to the client from a static root public folder.

The public root folder is a method commonly used to control client access to server folders, and can be set in Max by supplying the relative path to the user's Max patch as an argument to the hfmt.drawsocket object. This system allows users to organize their project in a mobile way, easily moved or installed on a new system.

Within the hfmt.drawsocket abstraction there is a helper script called *startscript.js* which retrieves the user patcher's folder path, and passes the path information as an argument to the node.script object on startup. [21] By default the folder containing the user's patch is used as the root public folder, however, users may wish to choose a different folder. For example, by setting the path "public_html", DRAWSOCKET will expect a folder called "public_html" to be located in the same folder as the Max patch running the hfmt.drawsocket abstraction, and if found will use this folder as the public root folder.

In larger projects it is often convenient to sort assets into separate folders for images, sound files, etc. For example if the user wishes to load an image file called "foo.jpg" located in a */public_html/images* subfolder, they would refer to their file at the address */images/foo.jpg*.

3. DRAWSOCKET API

The DRAWSOCKET API has developed organically as features are gradually added to the system, and has been rewritten several times as new use contexts have arisen.

The API was initially designed in keeping with the conventional "message" format used in the Max environment, so that drawing commands could be easily adapted from commands used for other Max drawing objects such as LCD, or jit.gl.sketch.

A Max message is structured as an array, beginning with a selector string, followed by a list of values, which is interpreted by the receiver based on a preexisting schema. However, as more features were added, some complications arose in regard to the sequence ordering, and as a result an alternative object-oriented API was developed, which makes use of a key-value approach that has proven more extendable for the DRAWSOCKET system (discussed below in section 3.2).

Note that in the discussion below, we use the term *message* as a general purpose term for communication via message events, which could be in different formats (OSC messages, Max messages, JSON objects sent as messages, etc.).

URL address routing. All messages sent to the server are addressed to a URL, which is used by the server to route messages to the appropriate clients. Multiple clients may be logged into the same URL, in which case they would all receive the same drawing commands. For example if you had a group of violins all playing the same part you could have them log into the URL:

http://server.ip.address:port/violin1

Then to send messages to the violin 1 section, you would use the URL address */violin1*, in the same way you wold use an OSC address.

[18] https://greensock.com/gsap

[19] Currently, the script is configured to check every 5 seconds, but this may change depending on performance on a larger scale system

[20] The node project is located in the package's /code/node folder, and includes all scripts, and configuration files.

[21] Note that this requires the user patcher to be saved to disk first, so that it has a valid folder location.

To send to all clients, regardless of URL address, DRAWSOCKET provides usage of the OSC /* wildcard address.

Note that in OSC, /* matches one single address level, whereas the DRAWSOCKET server uses the wildcard address to match any *URL*, which may include multiple slashes. To avoid confusion with OSC convention, it is strongly recommended to use single level addresses (i.e. use */violin1*, not */violin/1* in cases of indexed sub-groupings).

Object references. DRAWSOCKET uses the Document Object Model (DOM)[7] *id* attribute as the primary mechanism for referencing individual client objects from the server. On creation, the client-side script logs a reference to new objects with their unique ID, in a set of associative arrays which can be used for fast object lookup by name. Through this method, objects may be referred to by ID, modified, styled, transformed, or removed.

SVG and HTML nodes use the provided unique identifier as the node's *id* attribute, as per the DOM standard, while other objects such as GSAP animation or Tone.js sound objects are not in the DOM, but logged in the DRAWSOCKET's internal object model.

3.1 List-oriented API

The original "Max style" list-oriented DRAWSOCKET API design used a bundle of individual OSC messages, each of which performed an action on the client system. The list-oriented API has now been replaced, however a discussion of this approach is valuable, since it illustrates a structural limitation that we encountered with this syntax approach.

In the list-oriented API, the OSC message address was used as a way to specify several functional layers at once, through concatenating together multiple values separated by slashes. The general address syntax was made of three main levels: (1) the client URL address, (2) a unique object ID, followed by (3) a command string specifying the process to execute on the client system.

The commands provided a streamlined way to create and modify elements on the client browser, using a curated set of parameters.

For example the following OSC bundle:

```
{
    /violin/foo/draw/rect : [100, 100, 25, 25],
    /violin/foo/style/stroke−width : 1
}
```

contains two messages prefixed by */violin* which indicates that the server should send these commands to all clients logged into the */violin* URL.

Once received on the client system, the script would parse the OSC address, separating the ID from the command string. Here, the ID is "foo" and the command string is "draw/rect".

For each OSC message, the value attached to address was parsed by the client script based on schema defined in the documentation. In the case of "draw/rect", the message's value would be interpreted as defining an SVG rectangle's *x*, *y*, *width*, and *height* values. The second message works in a similar way, except that rather than creating a new object, it adds an inline SVG/CSS *style* attribute to the object node, setting the *stroke-width* parameter to a value of 1.

Grouping. Things start to get a little more complex with the list-oriented approach when attempting to define SVG group objects and object definitions. In these cases child objects can be grouped together and manipulated as a single graphic object, while not requiring each child object to have a unique ID.

The first problem we encountered was when trying to include two objects of the same type within the object. In the first implementation, a sub-bundle of OSC messages was used to group elements together, however since no IDs were required, the following example fails:

```
/*/groupex/draw/group : {
    /text : [210, 210, ''hi''],
    /text : [310, 210, ''bye'']
}
```

It fails because in OSC you are allowed to have multiple messages with the same address, however in Max the OSC messages need to be first converted to Max Dictionary format to be passed to the node.script object, and Max Dictionaries do not allow duplicate addresses.

Another complication arose when trying to style individual objects within a group, since there is no unique identifier to reference for adding inline style tags, and this is not accessible from the list-oriented API syntax. To address these two issues, the list-oriented grouping syntax was adapted to use an array of objects (aka sub-bundles in OSC). For example:

```
/*/groupex/draw/group : [{
    /path : ''M200,200a30,90,0,0,0,0−60a30,30,0,0,0,0,60'',
    /style : ''fill: black''
}, {
    /text : [210, 210, ''hi''],
    /style : ''fill: red''
}]
```

In this case, the *style* message is bound to the *path* message by wrapping them together in an object. This solution led to a reevaluation of the DRAWSOCKET API syntax, and resulted in the development of the object-oriented based API.

3.2 Object-oriented API

While the list-oriented approach provides a compact, one-line syntax, the list format is also limited, in that the list requires a predefined schema for how the list can be interpreted, and which types of operations the list values may address. The list-based approach is thus less easily extendable, since adding a new value to the list requires adding a new step in the interpreting script. The main benefit of the list syntax is that its compactness makes it sometimes faster for rapid prototyping, however the object-oriented approach can be more easily expanded as we will show below, and additionally, the object-oriented approach is helpful since it is self-describing, emphasizing legibility by associating a parameter name with each value.

For example, whereas above we drew a rectangle with a list, such as:

```
/violin/foo/draw/rect : [100, 100, 25, 25]
```

The same rectangle could be drawn with the object-based API using the "svg" *key*, and an *val* containing one or more

objects to process. The *"new"* keyword notifies the client that it should create a new SVG element:

```
/violin : {
  /key : ''svg'',
  /val : {
      /new : ''rect'',
      /id : ''foo'',
      /x : 100,
      /y : 100,
      /width : 25,
      /height: 25
    }
}
```

In the object-based approach each variable now has a name associated with its value, telling us what the variable represents. The list-approach is a more concise, requiring less typing, however, when we consider further what the messages are representing in the context of the DRAWSOCKET system, the benefit of the object approach becomes clearer.

SVG is based on the Extensible Markup Language (XML) format, [22] and is designed as tree of *nodes*, each with a set of *attributes* which are defined as key-value pairs. By using the same attribute names within the DRAWSOCKET object API, the client script can then simply insert as few or many of the attributes as it receives, rather than needing a specific set of attributes, as with the list-based approach. Also, by staying close to the original SVG API, the user can refer to the SVG specification directly to figure out which attributes they can use, rather than needing to limit their control parameters to those setup in the list parsing schema. [23]

For example, extending the above example, here we create two new objects, a rectangle and a circle, by defining them in an array, and additionally assign a CSS *class* reference for each:

```
/violin : {
  /key : ''svg'',
  /val : [{
    /new : ''rect'',
    /id : ''foo'',
    /x : 100,
    /y : 100,
    /width : 25,
    /height: 25,
    /class : ''room''
  }, {
    /new : ''circle'',
    /id : ''bar'',
    /cx : 112,
    /cy : 112,
    /r : 5,
    /class : ''source'
  }]
}
```

Keywords. There are currently four reserved keywords used with *svg* objects: *new*, *style*, *parent*, and *child*.

On receiving an *svg* object (or array of objects), the client-side script iterates each element of the array, and checks if there is an already existing object with that *id* tag; if so, it selects that element from the DOM lookup table. Next, the script checks if there is a *new* message in the object; if so, it creates a new node, either replacing the element at the existing *id*, or creating a new node if not already existing, and then processes the rest of the object messages.

If the object already exists, and no *new* is found, DRAWSOCKET will use the values in the object to update the object attributes. For example:

```
/violin : {
    /key : ''svg'',
    /val : {
      /id : ''foo'',
      /width : 100
    }
}
```

will change the *width* attribute of the node "foo" without modifying any other attributes that may have already been set.

3.3 Parent and child elements

Appending child nodes to parent SVG element can be accomplished via the *parent* and *child* keywords.

The *child* keyword, is a high-level API helper function that assists the user in specifying one or more child nodes in a tree syntax. The value attached to this address will be inserted as a child of the parent node, for example the inner text of a <text>element, or a new node within a <g>the SVG grouping element tag.

Here is an example of a circle and line contained in new SVG group, called "noteline":

```
/violin : {
    /key : ''svg'',
    /val : {
        /new : ''g'',
        /id : ''noteline'',
        /x : 100,
        /y : 100,
        /child : [{
            /new : ''line'',
            /id : ''liney'',
            /x1 : 10,
            /y1 : 5,
            /x2 : 100,
            /y2 : 5,
            /style : {
                /stroke-width : 1
            }
        }, {
            /new : ''circle'',
            /id : ''circley'',
            /cx : 5,
            /cy : 5,
            /r : 5,
            /style : {
                /stroke-width : 2,
                /fill : ''none'',
                /stroke : ''black''
            }
        }]
    }
}
```

Nodes with a *parent* attribute are inserted as children of the node with the *id* specified by the *parent*, as long as the parent element is already existing in the DOM. If no *parent* element is specified, the node is inserted into the default SVG group "main-svg".

[22] https://www.w3.org/TR/xml/

[23] That said, we have not yet fully tested the entire SVG specification. We believe the object API provides access to everything, but there maybe some unaddressed aspects.

For example, the above tree syntax could also be written this way:

```
/violin : {
    /key : ''svg'',
    /val : [{
        /new : ''g'',
        /id : ''noteline'',
        /x : 100,
        /y : 100
    }, {
        /new : ''line'',
        /id : ''liney'',
        /parent : ''noteline'',
        /x1 : 10,
        /y1 : 5,
        /x2 : 100,
        /y2 : 5,
        /style : {
            /stroke-width : 1
        }
    }, {
        /new : ''circle'',
        /id : ''circley'',
        /parent : ''noteline'',
        /cx : 5,
        /cy : 5,
        /r : 5,
        /style : {
            /stroke-width : 2,
            /fill : ''none'',
            /stroke : ''black''
        }
    }]
}
```

3.4 SVG layer drawing contexts

In an SVG file, each object element is drawn in the same order as they are written in the file, from top to bottom, with the last element being drawn last, "on top" of any objects that may have been drawn in the same location. In the DRAWSOCKET system, the drawing sequence is set through the order of the object creation (using the *new* keyword).

Using the *parent* and *child* keywords, new nodes can be created and inserted as children of existing nodes. The order in which the child nodes are created, sets the drawing order of the nodes. Importantly, *editing* nodes (i.e. setting values without the use of the *new* keyword), does *not* change the drawing order. Similarly, inserting nodes does not change the drawing order of the parent nodes. This rule makes it possible to use SVG groups as drawing layer contexts, which maintain stacking order relative to each other.

As an illustration, let's say you would like to have three layers, a background, middle and overlay. You could create three new groups within the main SVG node, called "back", "main", and "overlay", in a specific drawing order, like this:

```
/violin : {
    /key : ''svg'',
    /val : [{
        /new : ''g'',
        /id : ''back''
    }, {
        /new : ''g'',
        /id : ''main''
    } , {
        /new : ''g'',
        /id : ''overlay''
    }]
}
```

You could then use the *parent* keyword to append nodes to the newly created groups. New nodes are draw above older nodes, but since the groups maintain their drawing order, you can use them as layers. In this example, the "overlay" layer, will always be drawn after the "back" and "main" layer-groups.

```
/* : {
        /key : ''svg'',
        /val : [{
            /parent : ''main'',
            /id : ''clef'',
            /new : ''text'',
            /child : ''&#xE050'',
            /class : ''bravura_text'',
            /x : 40,
            /y : 50
        }, {
            /parent : ''back'',
            /new : ''rect'',
            /id : ''rect'',
            /x : 5,
            /y : 5,
            /width : 100,
            /height : 100,
            /fill : ''red''
        }, {
            /parent : ''overlay'',
            /new : ''circle'',
            /id : ''circle'',
            /cx : 50,
            /cy : 50,
            /r : 10,
            /fill : ''blue''
        }]
}
```

Using this approach, multiple layers of SVG elements can be grouped together and manipulated (with some limitations as described in the SVG specification).

3.5 SVG CSS Styling.

The ability to dynamically apply CSS styling operations on SVG elements provides the user with an extremely flexible mechanism for composing, and manipulating the graphic layout. For most common DRAWSOCKET usages, a set of default layout properties are defined in the file *drawsocket-default.css*, which is loaded with the default HTML file (*drawsocket-page.html*). The linked stylesheet sets some defaults for SVG element types, for example *lines* have a default stroke width value so that they are visible by default. [24]

DRAWSOCKET also provides dynamic access to CSS rules, for which it is useful to understand the hierarchy of SVG style properties.

There are three levels of inheritance with SVG CSS styling:

(1) *presentation attributes*, set within the element, e.g.:

<rect fill="red" >;

(2) *stylesheet* definitions, loaded via an attached CSS stylesheet document, or within a <style> element in the HTML document; and

(3) *inline styling*, a snippet of CSS wrapped in a string and set in an elements's *style* attribute, e.g.:

<rect style="fill: red; stroke: 2" >.

[24] Note that this is not always desirable, for example when importing SVG files exported by a program like Adobe Illustrator, which assumes that there are no pre-exiting CSS rules in place. For these cases, DRAWSOCKET users can either override the defaults via a new CSS definition, or change the .css file by hand.

Each is overridden by the next: stylesheets override presentation attributes, and inline styles override all the others. [25]

Using CSS *class* selector syntax opens up many possibilities. For example, here is an example using the object-array syntax to set create two CSS classes: (1) ".notehead" which sets defaults for fill and stroke properties, as well as the radius value, *r* ; and (2) ".notehead.open", a sub-class of ".notehead" which overrides the fill property.

Following the *css* definitions, a new SVG circle object is created and configured with the "notehead open" class.

```
/violin : [{
   /key : ''css'',
   /val : [{
     /selector : ''.notehead'',
     /props : {
         /stroke : ''black'',
         /stroke-width : 2,
         /fill : ''black'',
         /r : 5
     }
   }, {
     /selector : ''.notehead.open'',
     /props : {
         /fill : ''none''
     }
   }],
 }, {
   /key : ''svg'',
   /val : {
      /new : ''circle'',
     /id : ''foo'',
     /class : ''notehead open'',
     /cx : 20,
     /cy : 20
   }
 }]
```

3.6 SVG import and library definitions

DRAWSOCKET provides access to several methods for importing and reusing fragments of SVG. This is a useful approach for reducing the amount of data that needs to be sent over the network, and can greatly simplify the construction of more complex notation situations.

Referencing SVG definitions. There are two node types in the SVG specification which allow the user to create prototypes of graphic elements, *defs* and *symbol*, which can be applied like a stamp via the *use* tag.

Within the DRAWSOCKET main SVG element there is an element group called *defs* which is not directly drawn to the screen, but is visible by using the browser's HTML element viewer tool. DRAWSOCKET uses the same drawing context syntax for the *defs* node, as it does for the other drawing layers.

For example, the following snippet makes a new SVG group object in the *defs*, called "noteline", which contains a line and a circle:

```
/violin : {
   /key : ''svg'',
   /val : {
     /parent : ''defs'',
     /new : ''g'',
     /id : ''noteline'',
     /child : [{
         /new : ''line'',
         /x1 : 10,
         /y1 : 10,
         /x2 : 100,
         /y2 : 10
      }, {
       /new : ''circle',
       /cx : 5,
       /cy : 5,
       /r : 5
     }]
   }
 }
```

Typically the user would send a library of definitions at the beginning of the piece, and then refer to the set of definitions as needed via the *use* SVG element and its *href* attribute, creating an internal reference to a given definition selected through its *id* attribute. [26]

For example, the following new SVG object "foo", references the "noteline" definition above, offset to x, y position $\{100, 100\}$:

```
/violin : {
   /key : ''svg'',
   /val : {
     /new : ''use'',
     /id : ''foo'',
     /href : ''#noteline'',
     /x : 100,
     /y : 100
   }
 }
```

Importing fragments. The *use-href* syntax approach can also be used to import elements from external SVG files stored in the public HTML folder, by adding the target object's *id* to the external file path. For example, to reference an object with the ID "boo" in an external file called "other.svg" that is located in the public subfolder called "media" you could use the following snippet:

```
/violin : {
   /key : ''svg'',
   /val : {
     /new : ''use'',
     /id : ''foo'',
     /href : ''/media/other.svg#boo''
   }
 }
```

If *x* or *y* attributes are set in the *use* node, the referenced object will be offset by the amount specified by the *use* attributes.

DRAWSOCKET also provides an additional option with the *href* attribute. If the *href* value is a list, the second value is non-zero, the script will find the original object's bounding box and offset so that it lies at the origin $\{0, 0\}$, and then applies the x, y values as a second operation. The benefit of this feature is that it allows you to coordinate positions of objects without needing to know their original position in the reference file.

[25] With one exception, stylesheet definitions with the *!important* tag will override inline styles.

[26] Note that for all selections we are using the HTML/CSS # sign to specify that the following string is an *id*.

3.7 PDF import

PDF files may be imported into DRAWSOCKET. For example, to load a PDF, storing it at the DRAWSOCKET ID "foo", setting its *x* position, *width* and setting it to display page 2:

```
/* : {
    /key : ''pdf'',
    /val : {
     /id : ''newpdf'',
     /href : ''/media/flint_piccolo_excerpt.pdf'',
     /width : 600,
     /x : 100,
     /page : 2
   }
}
```

3.8 Animation

While DRAWSOCKET objects may be animated using native CSS transitions and keyframes, the GSAP TweenMax and TimlineMax libraries were introduced to provide a much more convenient and cross-browser supported method. With the TweenMax library users can create a "tween" transition between the object's current position and current CSS property values, to another set of values over a given amount of time, using the TweenMax.to function via the *tween* DRAWSOCKET key. For example:

```
/violin : {
   /key : ''tween'',
   /val : {
     /id : ''aaa'',
     /target : ''#note'',
     /dur : 10,
     /vars : {
       /x : 100,
       /y : 100,
       /opacity : 0
     }
   }
}
```

moves the SVG object "note" to the xy position $\{100, 100\}$, and fades the opacity to zero over a course of 10 seconds. The *tween* is stored as object in the DRAWSOCKET script at the given ID (here "aaa"), and may be recalled at will (see the online documentation for more details). The CSS selector *target*, *dur* and *vars* are plugged directly into the argument fields for the TweenMax.to function. [27]

More complex animations can be implemented with the TimelineMax function, via the DRAWSOCKET *timeline* command, which is comprised of an array of tweens (which can also have different targets). As with the TweenMax.to function, an effort was made to make the encoding syntax as close to the native GSAP Timeline function as possible so users can refer to the GSAP documentation for full reference.

```
/violin : {
   /key : ''timeline'',
   /val : {
     /id : ''foo_line'',
     /init : {
         /paused : ''true'',
         /yoyo : ''true'',
         /repeat : 20
     },
     /tweens : [{
         /target : ''#bar'',
         /dur : 1,
         /vars : {
             /y : 270,
             /x : 100,
             /scaleX : ''200%'',
             /opacity : 1,
             /ease : ''linear''
         }
     }, {
         /target : ''#bar'',
         /dur : 2,
         /vars : {
              /y : 10,
              /x : 0,
              /scale : ''100%'',
              /opacity : 1,
              /ease : ''linear''
         }
     }]
   }
}
```

DRAWSOCKET provides the *cmd* keyword for tweens (and timelines of tweens) to start, stop, reset, reverse, etc.

Synchronization. All commands sent from the server are timestamped, which provides DRAWSOCKET with a mechanism to synchronize animations. Using the Timesync.js library, the client periodically asks the server for its current clock time and logs an offset value between the two clocks. [8] Then, whenever a new animation start request is received, the client checks the message's timestamp relative to the current client clock time minus the logged difference from the server time to get the corrected animation start time in terms of the server clock. The client then checks the duration of the animation (tween or timeline) to make sure it hasn't already missed the end time for the animation, if not, the client script starts the animation, fast-forwarding if necessary to compensate for network lag.

3.9 Sound

In addition to providing access to browser-based drawing tools, DRAWSOCKET also makes use of the Tone.js [9] WebAudio [28] Framework for browser-based sound production.

The Tone support library also adds a new keyword, *call* which expects an object containing a *method* and optional *args*. Additionally, a the *call* object may also contain a *then* object which can be used as a sequential *call*, applied to the return value form the parent method call.

For example, he we create a new Tone.Player, load an mp3 file, tell it to start looping playback, and call the *toMaster()* Tone.Player method:

[27] https://greensock.com/docs/TweenMax/TweenMax()

[28] https://www.w3.org/TR/webaudio/

```
/* : {
  /key : ''sound'',
  /val : {
    /new : ''Player'',
    /id : ''kick'',
    /vars : {
        /url : ''/media/808_mp3/kick1.mp3'',
        /autostart : ''true'',
        /loop : ''true''
    },
    /call : {
      /method : ''toMaster''
    }
  }
}
```

3.10 HTML5

DRAWSOCKET provides access to HTML nodes via the *html* tag.

For example, this loads a video:

```
/* : {
  /key : ''html'',
  /val : {
    /new : ''video'',
    /id : ''foo'',
    /child : {
        /new : ''source'',
        /type : ''video/mp4'',
        /src : ''somerandommovie.mp4''
    }
  }
}
```

Some HTML5 JS objects also support the *call* keyword.

For example, this starts playing the above video:

```
/* : {
  /key : ''html'',
  /val : {
    /id : ''foo'',
    /call : {
        /method : ''play''
    }
  }
}
```

3.11 User Interaction

Lastly, DRAWSOCKET also sends user interaction information back to the server, outputting the data into Max where it can be used to control other processes, through mouse and multi-touch event listeners, and via HTML textfield input forms. When the user's mouse or fingers move over the screen DRAWSOCKET reports the x, y position and the top-most graphic object under the fingers or cursor, and bound with the URL address. [29]

DRAWSOCKET also provides access to HTML text input fields. To create a text field, users first create a form with a default text prompt and then position the form by applying a CSS transform, or tween. When a client enters text into the text input field and hits enter or clicks outside of the form, the text is sent back to the server and output in Max in a similar fashion to mouse and multi-touch data.

User defined event callbacks. The main client-side processing function can also be invoked from a callback for handling user interaction, exposed to the global JS namespace as *drawsocket.input*. For example, the following snippet, which creates an SVG path object, and assigns an *onclick* callback function which triggers a sample playback when the client user clicks on the path object:

```
/* : {
  /key : ''svg'',
  /val : {
    /new : ''path'',
    /id : ''wow'',
    /style : {
        /fill : ''red''
    },
    /d : ''M100,100a30,30,0,0,0,0−60a30,30,0,0,0,0,60'',
    /onclick : ''drawsocket.input({
        key: 'sound',
        val: {
            id: 'kick',
            call: {
                method: 'restart'
            }
        }
    })''
  }
}
```

4. FUTURE WORK

DRAWSOCKET currently still considered "in development", that said, the system has already been used in several live performances, and appears to be fairly robust. As we prepare for the large-scale extension of the system to the St. Pauli Elbtunnel [10] we will have a good opportunity to fully stress-test the system.

One potential issue that we imagine could arise is with the caching system. Currently the caching routine is processed within the callback function that gets called when a new dictionary arrives from Max. On receiving a new dictionary, the server routes the data, sending packets to the appropriate clients, and then sends the packets to the cache system which unions the data with any nodes with a matching *id* (or *selector* in the case of CSS). There is a question about the scalability of this approach.

Node.js, like vanilla JS, uses an "event driven", "single threaded event loop model", which uses a queue of event callbacks which need to be processed asynchronously. However, it is possible to block the event loop[11] within a callback function, should the execution take too long. In particular, the JSON.parse and JSON.stringify operations are potentially expensive, with a complexity of $O(n)$; so, depending on the size of the incoming dictionary, this could significantly slow down the response of the server. In our testing so far, we have already noticed some issues with processing very large dictionaries arriving from Max, but we need to investigate further. It is possible that since the data is broadcasted before being sent to the caching system, that the blocking of the event loop will be less noticeable on the client-side, however, the responsiveness of the server will be reduced and this will likely effect the clock-synchronization routine, and could also in extreme cases result in a buildup of events to process in queue. To address this issue, we might look into storing the URL states in a separate database, which runs as a separate process.

[29] Currently there is no unique client identifier, i.e. all users on the same URL will send their user interaction data identified by the same URL address. A unique tagging system will likely be implemented at some point.

Acknowledgments

The authors would like to thank Jacob Sello for his detailed testing the system which pushed the development of many new features and design considerations. We would also like to acknowledge the Federal Ministry of Education and Research in Germany (BMBF), for their support of this research through the Innovative Hochschule initiative.

5. REFERENCES

[1] M. Wright, "Open Sound Control: an enabling technology for musical networking," *Organised Sound*, vol. 10, no. 3, pp. 193–200, 2005.

[2] A. Freed, D. DeFilippo, R. Gottfried, J. MacCallum, J. Lubow, D. Razo, and D. Wessel, "o.io: a Unified Communications Framework for Music, Intermedia and Cloud Interaction." in *ICMC*, 2014.

[3] J. MacCallum, R. Gottfried, I. Rostovtsev, J. Bresson, and A. Freed, "Dynamic Message-Oriented Middleware with Open Sound Control and Odot," in *Proceedings of the International Computer Music Conference (ICMC'15)*, Denton, TX, USA, 2015.

[4] A. Trockman, S. Zhou, C. Kästner, and B. Vasilescu, "Adding sparkle to social coding: an empirical study of repository badges in the NPM ecosystem," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 511–522.

[5] G. E. Krasner, S. T. Pope *et al.*, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System," *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

[6] M. Halpern, Y. Zhu, and V. J. Reddi, "Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 64–76.

[7] M. Champion, L. Wood, G. Nicol, S. B. Byrne, A. L. Hors, P. L. Hégaret, and J. Robie, "Document object model (DOM) level 3 core specification," W3C, W3C Recommendation, Apr. 2004, http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/.

[8] "timesync.js," accessed 2019-1-9. [Online]. Available: https://www.npmjs.com/package/timesync

[9] Y. Mann, "Interactive music with tone.js," in *Proceedings of the 1st annual Web Audio Conference*. Citeseer, 2015.

[10] R. Gottfried and G. Hajdu, "Massive networked music performance in the old elbe tunnel," in *2019. Proceedings of the International Conference on Technologies for Music Notation and Representation-TENOR2019. Melbourne*, 2019.

[11] node.js. (2019) Don't Block the Event Loop (or the Worker Pool). [Online]. Available: https://nodejs.org/en/docs/guides/dont-block-the-event-loop/