SmartReflect: A Modular Smart Mirror Application Platform

Derrick Gold, David Sollinger, and Indratmo
MacEwan University
Edmonton, Alberta T5J 4S2
Email: {goldd5,sollingerd}@mymacewan.ca and indratmo@macewan.ca

***Abstract*—A smart mirror is a device that functions as a mirror with additional capability of displaying multimedia data, such as text, images, and videos. This device allows users to access and interact with contextual information, such as weather data, seamlessly as part of their daily routine. In this project, we developed SmartReflect—a software platform for developing smart mirror applications. The main features of SmartReflect are threefold: (1) It is modular, lightweight, and extensible; (2) It allows developers to sidestep the sandboxed environment created by web browsers; and (3) It supports plugins written in any programming languages. These improvements alleviate the hardware and software limitations inherent with the use of web browsers as a primary scriptable display method. In this paper, we describe the design and implementation of SmartReflect and compare it with other similar platforms. We also discuss the potential uses and applications of smart mirrors with regard to the new capabilities that our platform provides.**

Keywords*—smart mirror, magic mirror, interactive mirror, augmented mirror, augmented reality.

I. Introduction

With the introduction of system on chip (SOC), such as the Raspberry Pi, the notion of creating "smart devices" is a relatively new craze that has taken over hobbyist communities. One currently popular project using a SOC is the development of smart mirrors [1]–[3]. A smart mirror is a mirror with "smart" capabilities much like how cell phones have become smart. That is, it is a display that looks and acts like a mirror, but has the capability of displaying multimedia data through the mirror glass as if the mirror was a screen on its own accord. The major appeal of a smart mirror is that its physical design embeds a computational device in an ordinary piece of furniture that can integrate seamlessly into a home or working environment [4].

A common approach to building a smart mirror is to use a pane of two-way glass, a monitor, a frame to hold the glass and monitor, and a web browser with JavaScript to provide the software features and drive the display. The main limitation of this setup is related to the use of a browser as the display's method of information presentation. A browser creates a sandbox for the code that runs within it, that is, all interactions and processes are isolated from other running processes and hardware interactions on the computer. Furthermore, web applications are typically driven through user events generated on a web page (e.g., mouse clicks). This feature poses limitations in smart mirror applications. First, user events cannot be generated naturally in a browser when one interacts with the browser as one would with a mirror. Second, a sandbox limits the use of external hardware to generate events based on typical user-mirror interaction. Third, only JavaScript runs natively in a browser.

Consequently, such smart mirror platforms are typically limited in the following ways. First, they are not truly modular. Plugin systems exist, but require JavaScript knowledge to enable, disable, or configure plugins. Second, they use server-side solutions geared for web sites and RESTful (representational state transfer) API (application programming interface). The limitations of a RESTful API are inherent by the fact that users typically have no way of generating events to obtain data or to specify where on the server to obtain the requested data via natural mirror interactions. Third, the platforms are not inclusive for all programmers and programming methodologies. Only JavaScript is supported, which is geared for event-driven programming. No solutions exist for supporting other programming languages with their vast libraries of features and user base, hence, fragmenting the potential pool of developers for extending smart mirror features.

That being said, a web browser is still a necessary feature for providing and displaying information, as it has built-in support for multiple media formats, such as text, images, and videos. The information presentation can be made interactive with JavaScript and is customizable with CSS (Cascading Style Sheets). Furthermore, hyperlinking and web connectivity allows for borrowing and sharing of resources.

We designed and developed SmartReflect—a smart mirror platform that offers three main benefits [5]. First, it is modular and extensible. Developers can add plugins to customize their smart mirror applications. Second, it utilizes a server design that allows one to sidestep a sandbox created by a web browser. In our prototype, we demonstrated this feature by enabling users to interact with a smart mirror through an external hardware interface. Third, it allows for plugins to be created in all programming languages. With these problems addressed, an extensible platform is attainable, allowing for growth in smart mirror application development.

II. Related Work

As an everyday object at home, mirrors have great potential to serve not only as a reflective surface, but also as an interactive display as part of a smart home environment. In home automation domain, a smart mirror is commonly used for displaying multimedia data, promoting healthy lifestyle,

978-1-5090-0996-1/16/$31.00

and controlling household appliances [6]–[11]. The main appeal of this approach is that people can access personalized information effortlessly while doing their daily activities, such as washing hands and brushing teeth. Smart mirrors can also be used to monitor and control home appliances, such as lighting, heating, and security cameras.

Smart mirrors usually allow users some customization. To provide a personalized service, a smart mirror needs to identify the user who is standing in front of it, so that it can access and display the user's personal information, such as his/her schedule, to-do list, and appointments. Automatic methods for recognizing users include face recognition [9], [11], [12], tag-based identification, biometric data, and personal belongings (e.g., toothbrush) [8]. To select the most appropriate method, one should consider the location of a smart mirror and its intended uses. For example, the use of a camera (for face recognition) may not be suitable for a smart mirror installed in a washroom due to privacy reasons.

Smart mirrors can respond to user commands. Interaction methods supported by existing smart mirror systems include touch [9], voice [6], [7], gestures [10], and physical widgets [8]. Each method has its own strengths and weaknesses. For example, speech recognition software may perform poorly in a noisy environment. Therefore, using voice commands in a public space may not work well due to its high noise level. Touch-based interface does not suffer from this problem. However, using touch may not be appropriate for a smart mirror in a washroom, where the intended users likely interact with the mirror with wet or dirty hands.

Compared to existing systems, SmartReflect is designed to be simple, lightweight, and extensible. It does not require a lot of computational resources and can run on a Raspberry Pi. Our platform allows users to interact with a smart mirror through two interfaces. First, users can use a touchscreen mounted on top of a Raspberry Pi to write notes or control the smart mirror through a menu on the screen (see Fig. 1). Second, users can also use their mobile devices to access a web interface to configure the available plugins (see Section IV-C).

III. Design of the Platform

SmartReflect is designed to meet the requirements for creating a real-time display device. Our platform follows the Model-View-Controller (MVC) design pattern. The Model refers to plugins that manage data to be displayed. A plugin may retrieve data from third-party data resources (e.g., weather data API). The View is the screen/mirror that displays the data. The Controller is the server component that controls the execution of each plugin.

This design decision allows for the separation of concerns and defines the user space for each plugin with regard to each other contributing to an overall modular design. Communications between plugins, the server, and plugin clients in the browser are all handled through the WebSocket protocol [13]. WebSocket allows for real-time transmission of data as opposed to traditional servers that have the overhead of establishing and closing connections for each unit of data transferred or requested. This allows for plugins and the server to make rapid and multiple API calls that are necessary for a real-time dynamic display.

Fig. 1. Diagram of our smart mirror prototype.

A. Plugins

Plugins are responsible for providing information to display, as well as determining how the information is displayed. Using our platform, people can customize what information to display in their smart mirrors by writing their own plugins. A plugin is comprised of these components:

- HTML File: An HTML (HyperText Markup Language) partial describing the plugin's display structure within its display container.
- CSS File(s): One or more CSS files containing styling for a plugin's display container.
- JavaScript File(s): One or more JavaScript files that may fetch information or control the behavior of a plugin's display container.
- Script: An external program that generates information, or controls the behavior of a plugin's client container that cannot otherwise be achieved using JavaScript.
- Web GUI (Graphical User Interface): An interface that allows users to customize plugin-specific settings.

There are three types of scripts that a plugin can call:

- *One Shot*: A script that, after being executed once, is no longer executed again until a plugin is disabled, enabled, or reloaded. For example, a plugin displaying the Internet Protocol (IP) address of a smart mirror only needs to

be executed once, as this IP address does not change constantly.
- *Periodic*: A script that is scheduled to execute every X seconds and that does not need to maintain its states between execution. Such a script should complete its execution before being called again. A plugin displaying the current time is an exemplary use for periodic scripts.
- *Persistent*: A script that runs autonomously as a child process. These scripts are responsible for connecting to the server on their own and are provided the necessary connecting details (port and WebSocket protocol) to do so. Once connected, the program executes until it is requested to stop by the server, and is responsible for receiving, requesting, and sending its own draw calls to manipulate the plugin display container it is allotted. This type of script is useful for complex programs that require time-sensitive information exchange, such as real-time hardware monitoring/polling. A persistent script eliminates the start-up process for subsequent executions by establishing a single connection at the program's initialization for the plugin's lifetime.

To create a simple plugin using our software platform, one only needs to create an HTML partial, a CSS file, and potentially a JavaScript file depending on the plugin's purpose. For situations where one needs the ability to access external hardware, to create and load persistent data, or to use other advanced programming language features, a developer can write a script that works conjointly with or replaces entirely the JavaScript file(s). We demonstrated this advanced feature by developing a plugin that allows users to write notes on the mounted touchscreen and place their notes in the mirror.

The design of our plugin system also offers some flexibility. Not all plugins require a display component; they can be only a script running in the background. Alternatively, a plugin can be all display orientated with no scripts generating information or modifying its behavior. These properties are defined in a configuration file found at the root of each plugin's encompassing folder (see Section III-C)

B. Display

Our platform uses a web browser to display information due to its built-in support for various media formats. In our software stack, the display is comprised of two user spaces: global and plugin. The global user space is what the server uses for injecting and removing JavaScript, CSS, and HTML resources for each plugin, as well as for creating a container for each plugin. The plugin user space is made available for each plugin to modify and update itself without interfering with other plugin spaces. These plugin user spaces are bound to an HTML 'div' container that is created for each plugin by the global user space when a plugin is loaded to the display.

With these two user spaces defined, there exists another separation of concern that makes plugin development and customization easier. For one advantage, plugins do not need to worry about loading themselves; the server will do that for them. As another advantage, with the global user space defined, there exists a generalized set of operations for manipulating each plugin's container that a plugin does not need to worry about implementing themselves. One practical operation that can be generalized is the positioning of plugin containers on the display. Users should be able to (and can) rearrange plugins as they see fit without having to reconfigure a plugin's specific settings. Another general operation is the ability to enable and disable a plugin. Not all available plugins need to be loaded all the time. Users can customize which plugins they wish to use without having to delete or uninstall anything. From a resource usage perspective, this setup allows for the complete removal of plugins from the browser as they are unloaded, helping to keep the memory usage low on our tiny computer and prevent the execution of unnecessary JavaScript for idle plugins.

The plugin user spaces allow for the customization and display of plugin specific details in their own allotted screen space. Without these containers, the global user space would not be able to easily handle general operations for each plugin, as well as preventing plugins from intruding or overwriting each other's displays. In this user space, an API of general draw calls exists in which plugins can use for modifying its own container's display properties (see Section IV).

C. Server

Typically, web browsers communicate with a server based on user-generated events (e.g., mouse clicks), and the server responds only when a request is made to it. As a result, several smart mirror platforms have plugins that are not truly modular. That is, they always exist in memory with flow control statements (if, while, switch, etc...) determining whether or not to execute the given code for one plugin or another. Customizing these plugins must happen prior to loading a web page, and future changes require page reloads, resulting in a fairly static system. As mentioned earlier, such smart mirror software solutions also limit their plugins to running as JavaScript, unless a backend server is involved to run other software.

To solve this problem, we implemented a *reverse-server* system, in which a server is capable of generating events (i.e., passing draw calls) from an external plugin to drive a web browser display with or without the need for user-generated interactions. This reverse-server setup mimics the model that the X Window System [14] follows—where an "XServer" runs in the background with applications making X window calls—with the benefits of having multimedia and scripting capabilities of a web browser at our disposal.

The server component in SmartReflect has three main responsibilities: (a) Manage plugins and their dependencies; (b) Schedule and execute plugins and their scripts as necessary; and (c) Establish communications between the Plugin Client API and the Server API to enable external control for server and plugin features.

Plugins and their dependencies are managed using a file system and stored in a specified plugin directory, which is scanned by the server. In this root plugin directory, each plugin has its own directory containing its configuration file and other dependent files. The name of a plugin is determined by the name of the directory containing its plugin configuration file. Using the file system to manage plugins allows for easy installation and uninstallation of plugins. One simply copies or deletes a plugin directory to/from the plugin root directory. The file system also ensures that no plugin names are duplicated

because it is not permitted to have two directories with the same name in the same directory as each other.

Plugin scheduling occurs only when a plugin has a pending external script to execute. This scheduler operates on a timer that is configured on a per plugin basis. Two types of scheduling are possible: one-shot and periodic. One-shot scheduling adds a plugin's script to the scheduler, executes it once, and then removes the script from the scheduler, preventing it from being executed again. This type of script is useful for grabbing data that cannot be predicted, but once obtained, does not change during the run-time of the mirror (e.g., hardware information like an IP address). A one-shot script can only be executed again if the plugin is disabled and then enabled. A periodic script is executed every time the scheduled time interval has elapsed and never leaves the scheduler while the plugin is enabled. Data generated by a periodic script updates over time, predictably or not, and allows the server to update information on its own accord rather than having to create an persistent plugin script to do the same.

Communications between plugins and the display are handled through the WebSocket protocol [13]. The server is responsible for establishing this connection to allow Plugin Client API calls to be made without interacting with other running plugins. With this setup, plugins that require to make Plugin Client API calls do not need to establish themselves as a server to connect to the plugin clients. Instead, these plugins act as a client and send API calls to the server, which then forwards those calls to the appropriate plugin client and forwards plugin client responses back to the plugin (see Fig. 2). All of the networking is handled through a single port of entry and one server, rather than creating multiple servers for each plugin. This setup eases the deployment of the smart mirror software, utilizes less hardware resources, and allows for an all encompassing API to be created for managing plugins (see Section IV-C for more details).

IV. Application Programming Interfaces

To facilitate communications among plugins, display, and server in our platform, we developed a set of APIs.

A. Plugin Client API

The Plugin Client API is used by the server and *persistent* plugin scripts for sending and retrieving information to the plugin client container in a web browser (see Fig. 2). This API contains a reduced set of functions encompassing the most common operations necessary for dynamically updating web pages, and more specifically, updating the plugin client 'div' container on the display page. These instructions include the ability for sending text and HTML strings; getting, clearing, and setting CSS attributes; and executing JavaScript functions. Through this API, developers can dynamically update their plugin's presentation, add and retrieve textual contents from their plugin's client, and modify their plugin's behavior to create interactive and configurable plugins.

In contrast, plugins with *one-shot* or *periodic* scripts must have their styling defined prior to their execution, and their behavior is strictly limited to updating textual information (see Fig. 3). Plugins that have their own supplied JavaScript files and that have no external scripts do not need to utilize this

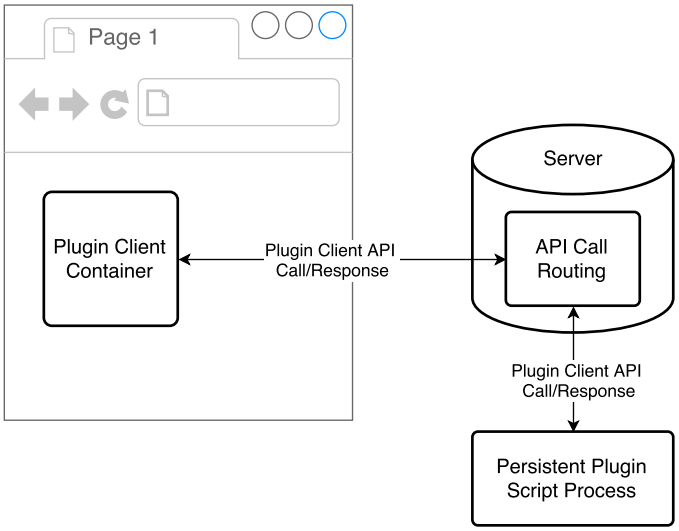

Fig. 2. Persistent scripts have bi-directional communications with their associated plugin's client. These scripts can send information and control plugin client behavior through CSS manipulation and JavaScript function calls defined in a plugin's JavaScript class object.

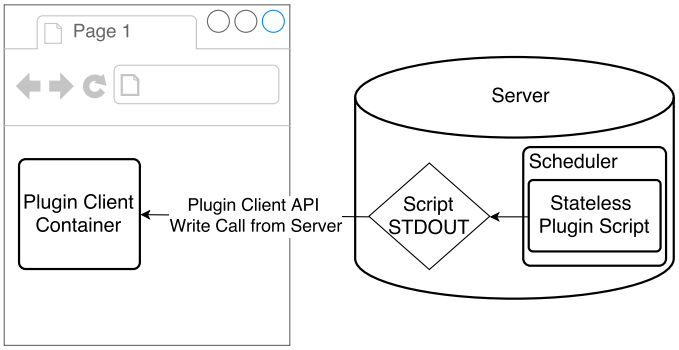

Fig. 3. One-shot and periodic plugin scripts have only one-way communications to their associated plugin's client. These scripts can only update textual contents on the display.

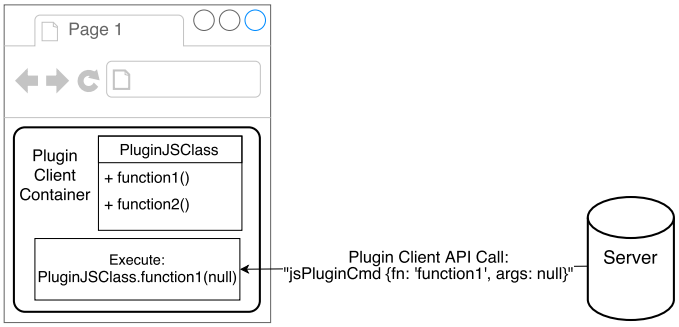

Fig. 4. Illustration of how to extend the Plugin Client API and call a new function that is defined in the extended API.

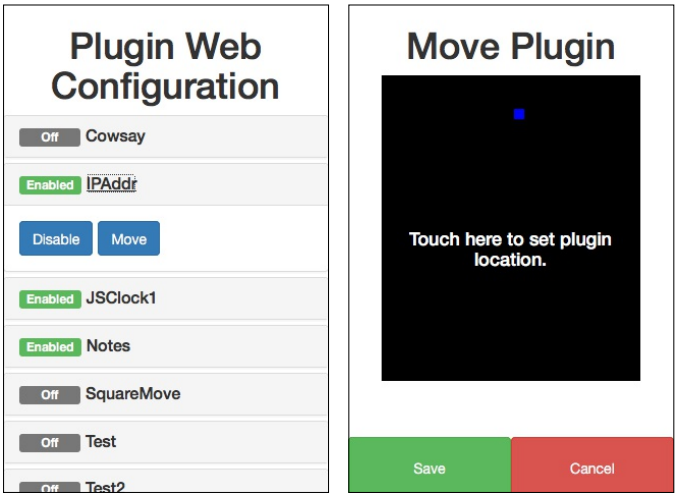

Fig. 5. The web management interface allows users to enable, disable, and move plugins using a web browser.

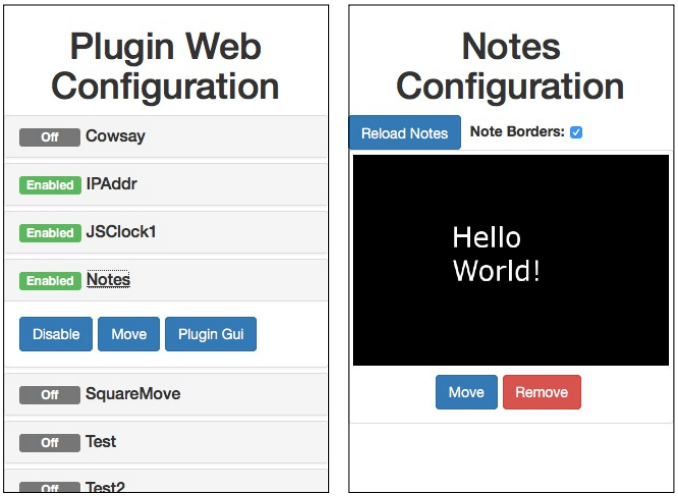

Fig. 6. A plugin may provide its own web GUI ('Plugin Gui'), which is accessible through the WMI, to support plugin-specific settings. In this example, the 'Notes' plugin-specific settings are shown.

API because such plugins can define their dynamic styling and behavior in their JavaScript programs.

Developers have the ability to extend this API by creating a JavaScript file containing a class object and using it conjointly with persistent scripts (see Fig. 4). This class object is instantiated when the display loads the plugin into the browser, and contains supplemental data structures and functions that do not exist in the reduced API instruction set. Through the Plugin Client API, developers can then call a function defined in this class object. With this implementation, developers have the ability to leverage the browser to execute complex operations rather than trying to recreate them with a series of API calls, thereby reducing the total number of calls overall.

B. Display API

The Display API is used by the server for loading and unloading plugins, and retrieving browser details. This API encompasses the global user space of the display, and is not accessible in the plugin user space.

C. Server API and Web Management Interface

The Server API is an all encompassing API that can make both Plugin Client and Display API calls, and that can set and get server configuration information. Through this API, we implemented a web management tool for enabling, disabling, and reloading plugins, getting plugin statuses, and modifying plugin settings on-the-fly. Through the web management interface (WMI), users can configure their mirrors using any browser, with particular interest in browsers on smart phones. With mobile browsers, users can use the touch display on their phones to arrange plugins with ease. The generic settings for customization—enable, disable, and move plugin—are available by default for each plugin (see Fig. 5).

Plugins may implement their own web GUIs for plugin-specific configuration (see Fig. 6). These GUIs are accessible through the WMI. When users configure such plugins, the plugins' web GUIs take control of the Server API and configure themselves as necessary. Having both a generic WMI and individual plugin web GUIs allows plugin developers to add configuration options that the generic WMI may not be aware of. As a result, plugin developers are not limited to the number of and types of settings.

A command line tool was also created to access the Server API. This tool allows for server administrators to script any processes necessary for configuring the server or plugins, as well as providing another method for other programs to access the API without needing to introduce a WebSocket library into the code base.

V. Prototype Implementation

This section describes the hardware and software implementation used to create our smart mirror prototype.

A. Hardware

We followed typical smart mirror building instructions to implement our prototype. As shown in Fig. 1, a pane of glass with a mirror film on one side is encased in a frame and placed on top of a monitor. The mirror acts in a similar way that a one way mirror works. When there is nothing displayed on the monitor (i.e., the monitor is black), users can see their own reflection in the mirror. When a non-black color is displayed on the monitor, that color appears to come through the glass from the monitor. We made a modification to the typical smart mirror design by adding a small 3.5-inch touchscreen on top of our Raspberry Pi. To best utilize this touchscreen, we have mounted the unit to the bottom side of the mirror. This modification was made to demonstrate the capability of SmartReflect to access external hardware interfaces—a task that is currently not possible with other similar smart mirror platforms that use a web browser as the display method.

B. Example of Plugins

SmartReflect is designed to be modular. Developers can write and add new plugins to the system easily. They just need to create a directory under 'Plugins' and save the necessary

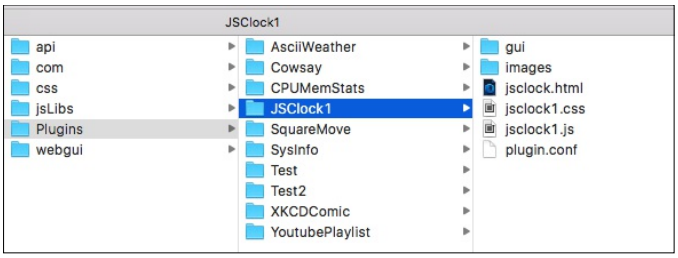

Fig. 7. Screenshot of how plugins are organized in the file system. Each plugin is stored in a separate directory and identified by the directory name. A plugin's directory contains all of the necessary files to run the plugin.

files in that directory (see Fig. 7). And to remove a plugin, they only need to remove the plugin's directory from the 'Plugins' directory. The platform uses directory names to identify the available plugins.

Below is an example of a one-shot plugin for displaying the IP address of a Raspberry Pi that powers a smart mirror. There are three main components of this plugin: a Bash script, a CSS file, and a 'plugin.conf' file. The Bash script is responsible for grabbing the IP address, wrapping text to display inside HTML 'div' elements, and sending the text to standard output.

```
#!/bin/bash
#sysinfo.sh
#Display Raspberry Pi IP Address on mirror.

#Get hardware interface info for any wireless
#adapters connected.
iface=$(ifconfig wlan0)

#If no wireless adapters are connected, get
#the ethernet port information.
if [ $? -ne 0 ]; then
  iface=$(ifconfig eth0)
fi

#If no information for the network interface
#could be retrieved, report the error to
#standard output for the mirror to display.
if [ $? -ne 0 ]; then
  echo "Error grabbing IP Address."
  exit 1
fi

#Otherwise, collect the IP address from the
#interface information collected from above.
ipaddr=$(echo "iface" | egrep -o
    "addr:([0-9]{1,3}[\.]?){4}" | sed
    's/addr://g')

#Print out the IP address in formatted HTML to
#standard output for the mirror to display.
echo '<div id="infoTitle">System IP
    Address</div>'
echo '<div class="infoParam">Ip
    Address:</div> <div
    class="infoValue">'$ipaddr'</div>'
```

The CSS file defines a set of rules that control the presentation of the Bash script's output in the mirror.

```
/*sysinfo.css*/

#SysInfo {
  position: absolute;
  left: 0;
  bottom: 0;
  width: 200px;
  height: 200px;
}

#infoTitle {
  font-size: 16pt;
}

#SysInfo .infoParam, #SysInfo .infoValue {
  font-size: 12pt;
  display: inline-block;
}

#SysInfo .infoParam {
  margin-left: 20px;
}
```

Finally, the 'plugin.conf' file contains all necessary information that is required by the server to run this plugin. In this example, the file specifies the paths of the CSS file and the Bash script, the type of this plugin (i.e., one-shot), and a flag to start this plugin on loading. The server differentiates between one-shot and periodic scripts by the 'script-timer' property in a script's 'plugin.conf' file. A negative integer indicates the script needs to be executed only once, where a positive integer keeps the script in the scheduler for later execution. Otherwise, both one-shot and periodic scripts are configured the same.

```
#plugin.conf

css-path:0="sysinfo.css"
script-path="sysinfo.sh"
script-timer=-1
start-on-load=true
```

This example shows the extensibility of SmartReflect and the simplicity of developing a new plugin. For a simple plugin, developers only need to write an HTML partial to standard output, provide a CSS file, and specify settings for the plugin. As illustrated in Fig. 3, the server will take care of the scheduling, processing, and displaying the plugin in the mirror.

VI. Potential Applications of Smart Mirrors

Smart mirrors have many potential applications in both personal and social settings. In personal settings, smart mirrors can be used to display relevant information, control household appliances, and provide emotional support to users [4], [6]–[12]. In bathrooms, smart mirrors could prove to be a valuable application for many people. As people prepare for their day, their hands are typically busy, but they typically stand in one spot performing low cognitive tasks. People could have their email messages, trending tweets or Facebook posts, and breaking news show up on their smart mirrors seamlessly.

Smart mirrors can facilitate communication within a family. Many homes have a mirror in their entrance or foyer. A smart mirror in this location could act as a central hub for family scheduling, which would increase awareness of each other's

activities within the family. Built right into the mirror, a digital calendar can be synced with users' calendar on their phone. Digital notes left for other users would not fall off from people walking by or opening the front door.

In public locations, smart mirror applications would be generalized, perhaps acting as conduits for information much like information kiosks, but provide physical and design benefits over implementing a kiosk. For example, there is limited space in mall washrooms to place a kiosk. A smart mirror, however, could display kiosk information, advertisements, and emergency alerts, all while utilizing the space that is already allocated for mirrors. Installed in fitting rooms, a smart mirror could revolutionize clothes shopping. No longer would someone need to physically try on all the clothes they are interested in. Using augmented reality, people could view themselves in digital versions of the available store wardrobe or experiment with various design options [15]. Such system would allow customers to browse a clothes catalog at a faster rate, filter down their selection of clothes, and only try on clothes to measure fit and comfort. The fitting room itself provides an excellent, low distraction environment for the processing of augmented reality, and could facilitate sensors for determining the correct size for customers.

VII. Conclusion and Future Work

Smart mirrors have great potential to enhance user experience of accessing and interacting with information. Not only do they allow users to see relevant information effortlessly, they can also be integrated into a larger system, such as a home automation system. It is quite common for smart mirror platforms to use web browsers as the primary display method, as web browsers offer built-in support for various media formats. However, such platforms are usually limited by the sandboxed environment that is created by web browsers.

To alleviate this problem, we developed a software platform for smart mirrors that offers the following benefits. First, our platform is designed to be lightweight. It runs on a tiny computer, such as the Raspberry Pi. Second, it is modular and extensible. Developers can implement their own plugins using any programming languages and integrate them to their smart mirror systems easily. Third, the server component in our platform enables a continous, real-time connection and remote draw calls. With this type of server configuration, native applications can be made with the capability of utilizing external, non-native hardware in the sandboxed environment of a web browser, while still utilizing the scriptable and multimedia rendering abilities that a browser provides.

As for future work, many new plugin opportunities are now available with the ability to access external hardware. It would be nice to explore various plugin ideas using motion detectors, temperature and light sensors, gesture recognition, voice commands, and some form of proximity detection, such as detecting the closest phone in range.

There are features that can improve the usability of SmartReflect. One feature would be the ability to install and uninstall plugins through the Server API. Currently, to install a new plugin, users have to login to their smart mirror servers and put the plugin's files to the plugin root directory. If the Server API supports the installation and removal of a plugin, we can use these features and develop a web GUI, so that users do not have to install/uninstall plugins manually.

Another enhancement would be to add a priority system for processing API calls. Currently, with wireless internet and network interference, there can be a delay in sending API calls. This delay causes the API calls to get cached on the server and then executed as the call queue is processed. With a priority system in place, non-important API calls could be dropped completely as to clear up the API call queue faster and try to alleviate the delay. Having prioritized API calls would benefit interactive processes that require real-time user input/output, such as rearranging a plugin's location in the mirror.

Finally, research into the appropriate uses of a smart mirror is warranted. Mirror usage usually involves a local presence, with motion and visual interaction, and depends on the context and location of the mirror. Some locations may allow for more personal applications, while others to more generalized purposes for a large user base. People may develop different mental models about how a smart mirror works, which may shape their behavior while interacting with it and raise issues, such as privacy and security. Understanding of user needs and expectations is key to successful deployment of smart mirrors.

References

[1] E. Cohen, "Smart mirror," http://smart-mirror.io/, accessed: 2016-07-14.

[2] H. Mittelstaedt, "HomeMirror," https://github.com/HannahMitt/HomeMirror, accessed: 2016-07-14.

[3] M. Teeuw, "Xonay labs," http://michaelteeuw.nl/tagged/magicmirror, accessed: 2016-07-14.

[4] H. Sukeda, Y. Horry, Y. Maruyama, and T. Hoshino, "Information-accessing furniture to make our everyday lives more comfortable," *IEEE Transactions on Consumer Electronics*, vol. 52, no. 1, pp. 173–178, 2006.

[5] D. Gold and D. Sollinger, "Smart reflect server," https://github.com/DerrickGold/SmartReflectServer, accessed: 2016-07-30.

[6] L. Ceccaroni and X. Verdaguer, "Magical mirror: Multimedia, interactive services in home automation," in *Proceedings of the Workshop on Environments for Personalized Information Access*, 2004, pp. 10–21.

[7] J. R. Ding, C. L. Huang, J. K. Lin, J. F. Yang, and C. H. Wu, "Interactive multimedia mirror system design," *IEEE Transactions on Consumer Electronics*, vol. 54, no. 3, pp. 972–980, 2008.

[8] K. Fujinami, F. Kawsar, and T. Nakajima, "AwareMirror: A personalized display using a mirror," in *Proceedings of the Third International Conference on Pervasive Computing*, 2005, pp. 315–332.

[9] M. A. Hossain, P. K. Atrey, and A. E. Saddik, "Smart mirror for ambient home environment," in *Proceedings of the Third IET International Conference on Intelligent Environments*, 2007, pp. 589–596.

[10] T. Lashina, "Intelligent bathroom," in *Proceedings of the Workshop on Ambient Intelligent Technologies for Wellbeing at Home*, 2004.

[11] C. Sethukkarasi, V. S. HariKrishnan, and R. Pitchiah, "Design and development of interactive mirror for aware home," in *Proceedings of the First International Conference on Smart Systems, Devices and Technologies*, 2012, pp. 1–8.

[12] Y. C. Yu, S. D. You, and D. R. Tsai, "Magic mirror table for social-emotion alleviation in the smart home," *IEEE Transactions on Consumer Electronics*, vol. 58, no. 1, pp. 126–131, 2012.

[13] Internet Engineering Task Force (IETF), "The WebSocket Protocol," https://tools.ietf.org/html/rfc6455, accessed: 2016-07-16.

[14] R. W. Scheifler and J. Gettys, "The x window system," *ACM Trans. Graph.*, vol. 5, no. 2, pp. 79–109, 1986.

[15] D. Saakes, H.-S. Yeo, S.-T. Noh, G. Han, and W. Woo, "Mirror mirror: An on-body t-shirt design system," in *Proceedings of the Conference on Human Factors in Computing Systems*, 2016, pp. 6058–6063.