**Chapter 1**

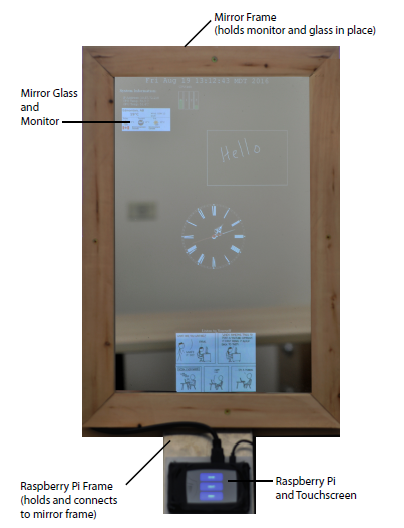
**INTRODUCTION**

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. 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.

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 en-able, 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 methodolo-gies. 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.



**Fig 1. Diagram of our smart mirror prototype.**

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. First, it is modular and extensible. Developers can add plug-ins 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 plug-ins to be created in all programming languages in our case we use Python as the Programming Language. With these problems addressed, an extensible platform is attainable, allowing for growth in smart mirror application development.

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, and controlling household appliances. The main ap-peal 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 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, tag-based identification, biometric data, and personal belongings (e.g., toothbrush). 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.

**Chapter 2**

**LITERATURE SURVEY**

An advanced information society is now arising with the rapid progress of information technology. Information equipment, such as personal computers, mobile phones, and large-sized displays are essential to our daily life, and they enable us to seamlessly access various kinds of information anytime and anywhere.

1. **Smart Mirror for Ambient Home Environment.**

**Authors:** M. Anwar Hossain, Pradeep K. Atrey and Abdulmotaleb El Saddik

This paper describes the design and development of a futuristic smart mirror that represents an unobtrusive interface for the ambient home environment. The mirror provides a natural means of interaction through which the residents can control the household smart appliances and access personalized services. Emphasis is also given to ensure convenience in accessing these services with a minimum amount of user intervention. For example, face recognition-based authentication is used to automatically identify the user facing the mirror and provide widget-based interface to access data feeds and other services. Aservice-oriented architecture has been adopted to develop and deploy the various services, where the mirror interface, the appliances, and the news and data feeds all use web service communication mechanisms. The smart mirror functionalities have been demonstrated bydeveloping an easily extendable home automation system that facilitates the integration of household appliances and various customized information services.

1. **Information-Accessing Furniture To Make Our Everyday Lives More**

**Comfortable.**

**Authors:** Hiroko Sukeda, Youichi Horry, Yukinobu Maruyama, and Takeshi Hoshino

We present a concept for information equipment called information-accessing furniture. We have developed an embedded module to be assembled into tables, mirrors, and walls. It can be easily assembled into furniture or other interior components because it is a package of information processing units, input/output units, and other optional units. This concept enables building information equipment designed to match a surrounding environment, and offers an intuitive interface to users, thus helping them to obtain information easily while doing routine activities. We developed several different prototypes with embedded modules and studied their applications. Some comments we received during interviews with end-users and interior designers are also presented.

1. **Interactive Multimedia Mirror System Design.**

**Authors:** Jun-Ren Ding, Chien-Lin Huang, Jin-Kun Lin, Jar-Ferr Yang.

This investigation describes a novel design and implementation of an interactive multimedia mirror system, called “magic mirror.” The magic mirror implemented in a personal computer is equipped with normal peripherals such as CCD camera, infrared ray devices, microphone, speakers, and an LCD monitor, which is further covered with a piece of general reflective glass. The magic mirror integrates self-developed software, which includes speech recognition, speech synthesis, video detection, and 3D graphics to provide easy video and audio interactions. With Internet connections, the magic mirror can capture the instant weather, stock information, multimedia sources, etc. By verbal comments, the users can easily activate personal multimedia services to provide visual information on display and verbal message from graphical genius. The magic mirror with emotion speech recognition and synthesis can be acted as a friendly agent, who can listen to your questions and automatically response the requests for information, relaxation, and consolation. Besides, the magic mirror with video detection can perceive the human gestures and behaviors to achieve possible visual interactions.

1. **Smart Mirror: A Novel Framework for Interactive Display.**

**Authors:**Athira S, Frangly Francis, Radwin Raphel, Sachin N S, Snophy Porinchu, Ms.Seenia Francis.

Our lifestyle has evolved in such a way that optimizing time is the most important thing. Based on the user studies and prototype implementation, we present the development of an innovating appliance that incorporates interactive services of information, offered through a user interface on the surface of a mirror. Our work is based on the idea that we all looks at the mirror when we go out, so why wouldn’t the mirror become smart. The framework will offer basic services, like the presentation of personalized weather data, time, date and will incorporate some additional functionality, like reminder service by mobile synchronization and through social media. Our framework is based on detecting presence of human using Passive Infrared sensors and Wi-Fi connectivity. Once a person comes in front of the mirror, it displays the information that is being fed from the web. This data or information includes calendar, time, weather, news feed, notifications and so on. Our framework also discusses about the speech recognition and its application in control mechanism in home appliances and opening and closing of shelf. We use speech recognition to automate many tasks that usually requires hands-on human interaction, such as recognizing spoken commands to perform something like turning on lights or shutting a door. Our framework also introduces speech activated music player, and plays the music when a person gives a command.

**Chapter 3**

**PROBLEM STATEMENT**

Smart mirrors can respond to user commands. Interaction methods supported by existing smart mirror systems include touch, voice, gestures, and physical widgets. 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.

**3.1 Problem Statements**

Previously, Voice Recognition and Touch-Based Smart Mirrors required a lot of computational resources.

There was no remote access to the Web Interface to configure the data which is displayed in the form of plug-in.

Previously used processors lacked extensibility and were hefty. On the other hand Raspberry Pi is a Light-Weight and extensible computing device.

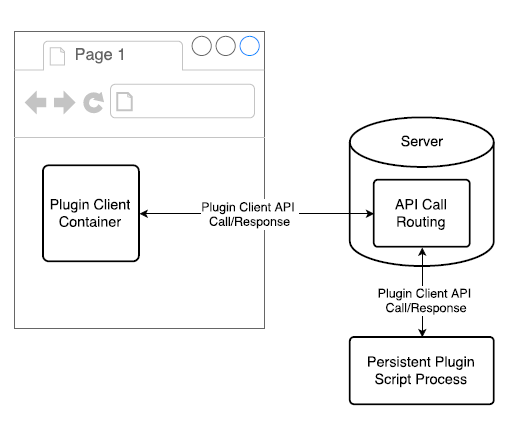
**3.2 Existing System**

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.

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.



**Fig 2. Persistent scripts have bi-directional communications with their associated plugin’s client.**

**3.3 Proposed System**

The proposed Smart Mirror is modular and extensible. Developers can add plugins to customize their smart mirror applications.

Also it utilizes a server design that allows one to sidestep a sandbox created by a web browser.

In our System we demonstrated this feature by enabling users to interact with a smart mirror through an external hardware interface.

Interaction with external hardware interface allows for plug-in to be created in all programming languages.

* 1. **Advantages of Proposed System**
* Compared to existing systems, Smart Reflect is designed to be simple, lightweight, and extensible. It does not require a lot of computational resources and can run on a Raspberry Pi.

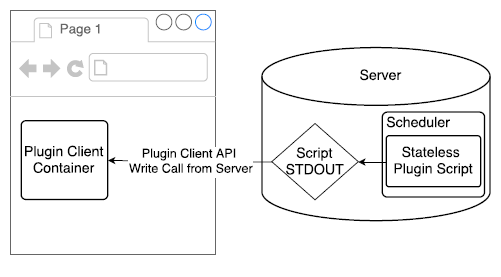
Our platform allows users to interact with a smart mirror through mobile devices to access a web interface to configure the available plug-in.

Our platform follows the Model-View-Controller (MVC) design pattern. The Model refers to plug-in that manage data to be displayed. A plug-in may retrieve data from third-party data resources.

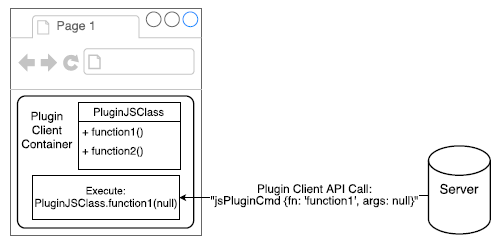
This design decision allows for the separation of concerns and defines the user space for each plug-in with regard to each other contributing to an overall modular design.

Communications between plug-in, the server, and plug-in clients in the browser are all handled through the WebSocket protocol . 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 plug-in and the server to make rapid and multiple API calls that are necessary for a real-time dynamic display.

****

**Fig 3. One-shot and periodic plugin scripts have only one-way communications to their associated plugin’s client.**

****

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

**Chapter 4**

**SYSTEM REQUIREMENTS SPECIFICATION**

A system requirements specification (SRS) is a description of a software system to be developed. It lays out functional and non-functional requirements, and maybe include a set of use cases that describe user interaction that the software must provide. The software requirement specification document enlists enough and necessary requirements that are required for the project development.

**4.1 USER ROLES:**

A user role defines permissions for users to perform a group of tasks. User roles also determine the access level or permissions of a person authorized to use the particular document.

There are two types of users for the Smart Mirror,

**PROGRAMMER:** A user proficient in Programming Languages who can modify, extend or develop the code which in our case is in Python Language.

**GENERIC USER:** A user who uses the mirror for everyday activities and takes full benefit of the extra features of the Smart Mirror without having to worry about the inner functionalities of the Smart Mirror.

**4.2 HARDWARE REQUIREMENTS**

Set of requirements defined by any operating system or software application in the physical computer resources is known as hardware requirements.

* Display Device : A wifi enabled Monitor.
* Reflective Surface : A two-way mirror, fit to the size of the display.
* Frame : Thin wood to build a mirror frame.
* Processor : Raspberry Pi.
* Input device : A keyboard for initial setup.
* Other : WiFi USB dongle, HDMI cord, power supply.

**4.3 SOFTWARE REQUIREMENTS**

Software requirements is a part of software engineering that deals with establishing the needs of the users that are to be solved by given software.

* Operating System : Linux (Raspberry Pi compatible).
* Coding Language : Python(Package).
* Web Server : Apache.
* Drivers : WiFi Drivers to Raspberry Pi.

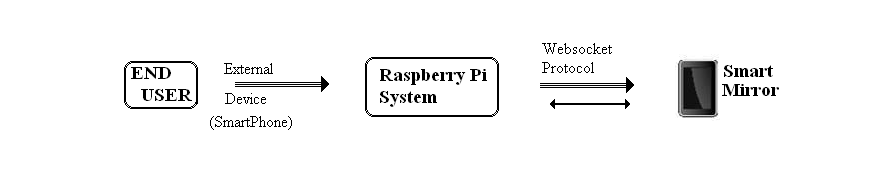
**Chapter 5**

**SYSTEM ANALYSIS AND DESIGN**

System Analysis and Design is an interdisciplinary part of science, which can be related to System analysis and System design. System analysis is a problem solving technique that decomposes a system into its component pieces for the purpose of the studying how well those component parts work and interact to accomplish their purpose or System analysis is the process of studying a procedure or business in order to identify its goals and purposes and create systems and procedures that will achieve them in an efficient way. Systems design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. Systems design could be seen as the application of systems theory to product development.

**5.1 System Architecture**

A system architecture or systems architecture is the conceptual model that defines the structure, behavior, and more views of a system. An architecture description is a formal description and representation of a system, organized in a way that supports reasoning about the structures and behaviors of the system.



Smart Mirror 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 plug-ins that manage data to be displayed. A plug ins 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 plug-in. This design decision allows for the separation of concerns and defines the user space for each plug-in with regard to each other contributing to an overall modular design.

Communications between plug-ins, the server, and plug in clients in the browser are all handled through the Web Socket protocol [13]. Web Socket 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 plug-in and the server to make rapid and multiple API calls that are necessary for a real-time dynamic display.

**5.2 Modules**

**Module 1: Plug-ins**

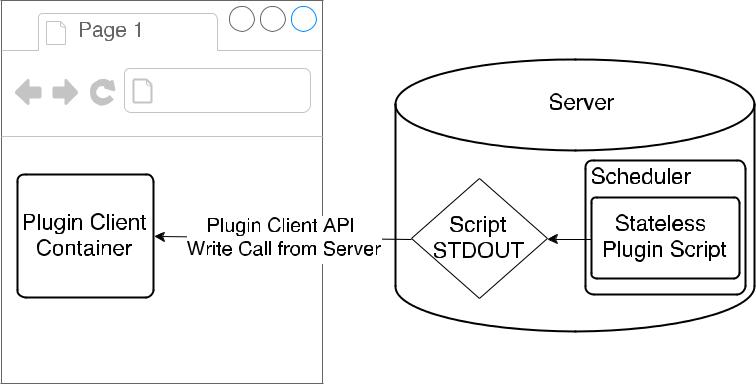
Plug-ins are responsible for providing information to display, as well as determining how the information is displayed. Using our platform, people can customize what information to display in their smart mirrors by writing their own plug-in.

. A plug-in is comprised of these components:

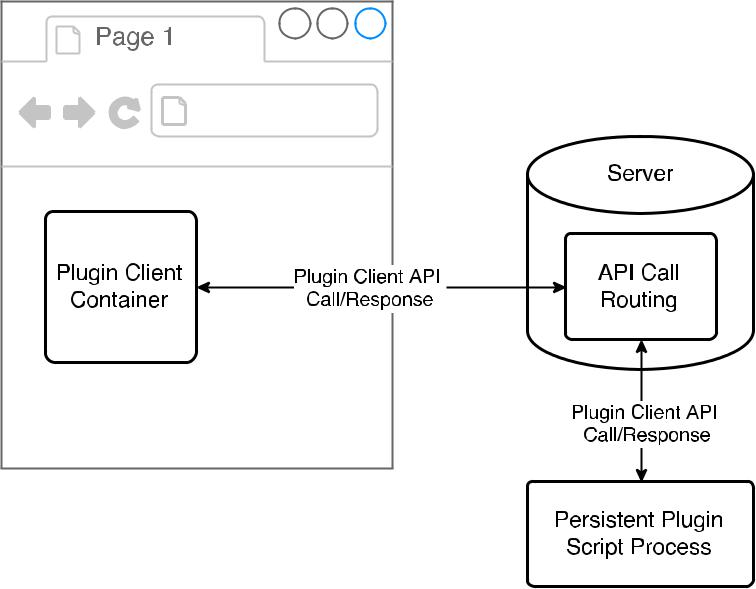
* **HTML File:** An HTML (HyperText Markup Language) partial describing the plugin’s display structure within its display container.
* **CSS File(s):** One or more CSS files containing styling for a plugin’s display container.
* **JavaScript File(s):** One or more JavaScript files that may fetch information 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 plug-in-specific settings.

There are three types of scripts that a plug-in can call:

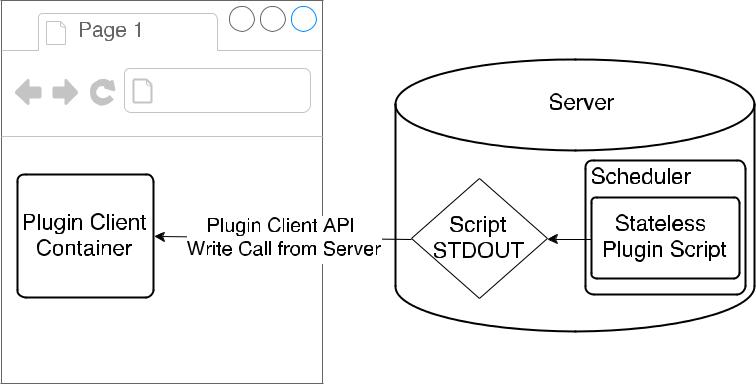
* **One Shot:** A script that, after being executed once, is no longer executed again until a plug-in is disabled, enabled, or reloaded. For example, a plug-in displaying the Internet Protocol (IP) address of a smart mirror only needs to be executed once, as this IP address does not change constantly.



* **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 plug-in 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 allot-ted. 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).

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.

**Module 2: Display**

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

With these two user spaces defined, there exists another separation of concern that makes plug-in development and customization easier. For one advantage, plug-ins 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 plug-in does not need to worry about implementing themselves. One practical operation that can be generalized is the positioning of plug-in containers on the display. Users should be able to (and can) rearrange plug-ins as they see fit without having to reconfigure a plugin’s specific settings. Another general operation is the ability to enable and disable a plug-in. Not all available plug-ins need to be loaded all the time. Users can customize which plug-ins they wish to use without having to delete or uninstall anything. From a resource usage perspective, this setup allows for the complete removal of plug-ins 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 plug-ins.

The plug in user spaces allow for the customization and display of plug-in 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 plug-in, as well as preventing plug-ins from intruding or overwriting each other’s displays. In this user space, an API of general draw calls exists in which plug-ins can use for modifying its own container’s display properties.

**Module 3: 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 plug-ins 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 plug-in or another. Customizing these plug-ins 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 plug-ins 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 plug-in 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:

* Manage plugins and their dependencies.
* Schedule and execute plugins and their scripts as necessary.
* 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 schedul-ing 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.

**5.3 Flow Chart**