Vulture: Cross-Device Web Experience with Fine-Grained Graphical User Interface Distribution

Seonghoon Park†, Jeho Lee†, Yonghun Choi‡, and Hojung Cha†*
†Department of Computer Science, Yonsei University, Seoul, Republic of Korea
‡Korea Institute of Science and Technology, Seoul, Republic of Korea
{park.s, jeholee}@yonsei.ac.kr, y.choi@kist.re.kr, hjcha@yonsei.ac.kr

***Abstract*—We propose a cross-device web solution, called Vulture, which distributes graphical user interface (GUI) elements of apps across multiple devices without requiring modifications of web apps or browsers. Several challenges should be resolved to achieve the goals. First, the peer–server configuration should be efficiently established to distribute web resources in cross-device web environments. Vulture exploits an in-browser virtual proxy that runs the web server's functionality in web browsers using a virtual HTTP scheme and a relevant API. Second, the functional consistency of web apps must be ensured in GUI-distributed environments. Vulture solves this challenge by providing a single-browser illusion with a two-tier document object models (DOM) architecture, which handles view state changes and user input seamlessly in cross-device environments. We implemented Vulture and extensively evaluated the system under various combinations of operating platforms, devices, and network capabilities while running 50 real web apps. The experiment results show that the proposed scheme provides functionally consistent cross-device web experiences by allowing fine-grained GUI distribution. We also confirmed that the in-browser virtual proxy reduces the GUI distribution time and the view change reproduction time by averages of 38.47% and 20.46%, respectively.**

***Index Terms*—cross-device experience, graphical user interface distribution, web applications, mobile web.**

I. Introduction

With the proliferation of diverse computing devices, such as smartphones, tablets, smart televisions (TVs), and others, many of these devices are commonly owned by individuals. In general, an application running on a device is often bound to operate on that single device. This limits the potential of exploiting multi-device environments for an enriched user experience. If the graphical user interface (GUI) of an application were allowed to be distributed to other devices as demanded by the user, the cross-device experience [1] in multi-device environments would be more versatile than what is found today. For instance, when a user is taking a note using the Google Keep application on a laptop computer, the user may wish to move a canvas-related GUI element to a tablet computer, intending to draw figures precisely with a stylus pen. Enabling this kind of functionality distribution in the application requires users to migrate a specific GUI component to a target device.

Unfortunately, most of the cross-device solutions developed thus far simply share the entire screen among the devices or cast specific types of GUI components without actually distributing the GUI elements to the devices at a fine-grained level. Screen mirroring (or screen sharing) solutions, such as VNC [2], AnyDesk [3], and Chrome Remote Desktop [4], mirror the entire screen of a host device to a target device. Chromecast [5] simply casts the video streams from a mobile device to a big screen, such as a smart TV. In short, existing solutions are not able to support the GUI distribution of an application at the component level, hence resulting in insufficient support for cross-device experiences. Recently, efforts have been made to provide sophisticated cross-device experiences beyond simple mirroring, especially for popular web apps such as YouTube and Google Docs. The approach, however, requires considerable effort to reauthor the original application to provide the functionality. Furthermore, this cross-device functionality is limited because the distribution of GUI components is preset by developers.

Research has been conducted to exploit the potential of a cross-device experience, especially one capable of providing fine-grained GUI distribution without reauthoring, for both native apps and web apps. In the case of native apps [6]–[8], cross-device GUI distribution is provided for devices running a homogeneous operating environment and, thus, is not applicable to heterogeneous platforms. Unlike native apps, web apps running on web browsers operate in a platform-independent fashion. Therefore, cross-device solutions based on web apps provide versatility on different platforms, require fewer efforts by developers, and ensure practicality because the widely available web apps are equivalent to native apps. One recent attempt was XDBrowser [9], which distributes the GUI of web apps to devices without reauthoring. The work primarily focused on the human–computer interaction (HCI) issue of distributing GUIs for the best possible views across multiple devices. However, enabling techniques for cross-device operations, such as the distribution, synchronization, and authorization of GUI elements, were not discussed. To provide a cross-device web experience, many of the practical issues in distributing GUIs across multiple devices, such as access rights handling, view state updates, and user input synchronization, should be solved. To date, no work has delivered cross-device web functionality that is fully functional without reauthoring applications.

Two key challenges exist in providing the desired cross-device web functionalities. The first challenge is to develop an efficient and effective peer–server scheme to distribute web resources. We define a *host* as a primary device on which a user selects the GUI elements to distribute to other devices called *peers*. Peers need web resources to render the GUI elements transmitted from a host. Peers often fail to fetch web resources directly from the original web servers because they are constrained by access rights. Proxy servers can be exploited to solve these issues, but privacy and cost issues exist. The second challenge is to ensure the functional consistency of web apps in GUI-distributed environments. The JavaScript runtimes in the browsers handle the core logic of web apps and interact with document object models (DOMs) that represent GUIs. The

* Corresponding author

interactions assume single-browser environments; thus, web browsers cannot guarantee the validity of interactions in cross-device environments. From the users' point of view, user inputs from multiple devices must be handled properly, and GUI states should be synchronized among the devices.

In this paper, we propose Vulture, a platform that *readily* provides cross-device experiences in web environments. Vulture provides novel solutions to the challenges above. To orchestrate a peer–server configuration, Vulture introduces an in-browser virtual proxy that efficiently handles the distribution of web resources for peers. The in-browser virtual proxy brings proxy servers into web browsers; thus, the approach mitigates the privacy and cost issues of proxy servers. Unfortunately, current web browsers do not provide hypertext transfer protocol (HTTP) sockets to web apps, making it difficult to implement the virtual proxy in web platforms. To solve this issue, we propose a cross-device virtual HTTP scheme and a relevant API for the virtual HTTP. This scheme practically empowers web browsers to run web servers. To ensure functional consistency for GUI-distributed web apps, Vulture provides a mechanism for single-browser illusions based on a two-tier DOM architecture, which consists of original and cloned DOMs. The original DOM is located in a host to handle the core logic of web apps, while the cloned DOM is located both on the host and peers to render the GUI elements. With this scheme, the host handles the execution of JavaScript and user inputs in their entirety. The peers simply receive the resulting view states from the host and forward the user inputs to the host.

To the best of our knowledge, Vulture is the first approach allowing for a mature level of cross-device experience in web environments without having to reauthor applications or modify browsers. The key contributions of our work are as follows:

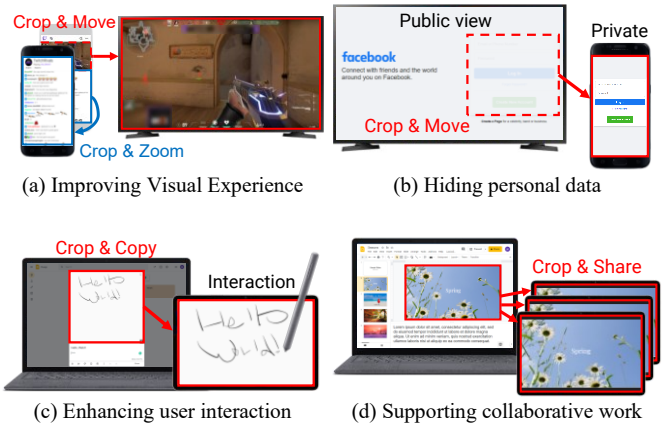
- Vulture introduces an in-browser virtual proxy to efficiently and effectively relay web resources from hosts to peers. To enable web servers to be executed in web browsers, we propose a cross-device virtual HTTP scheme and a relevant developer API.
- To ensure the functional consistency of GUI-distributed web apps, Vulture provides single-browser illusions by synchronizing view states and user inputs among multiple devices while allowing users to distribute GUIs at a fine-grained level.
- We validate the proposed scheme by demonstrating fully functional cross-device web experiences across a diverse set of heterogeneous devices operating on various real web apps.

II. Cross-Device Web Experience

We describe the practicality of the cross-device web experience provided by multiple devices running a distributed GUI in a web app. We classify the use case into four categories and describe the scenarios in detail. Fig. 1 illustrates the use case for each category.

A. Improving Visual Experience

When a web app contains GUI elements whose functionality can be distributed and run on multiple devices simultaneously, the user experience will be greatly enhanced or even maximized. For example, when chatting with people while watching a video on Twitch using a smartphone, a user may want to move the video-related GUI elements to a smart TV, which provides a large screen, while continuing to chat with the smartphone (C1.1)[1]. This cross-device web scenario will improve the user experience in the given situation. Another example is when searching for a place with the OpenStreetMap on a smartphone, the visible portion of the map on the smartphone screen is reduced because the search interface and virtual keyboard are put in the foreground (C1.2). If the map-related GUI elements are transmitted to another device that has a large screen, such as a tablet computer, the user experience would be much improved by viewing the map on the tablet while searching for places with the smartphone.



(a) Improving Visual Experience (b) Hiding personal data

(c) Enhancing user interaction (d) Supporting collaborative work

Fig. 1. Cross-device web experiences.

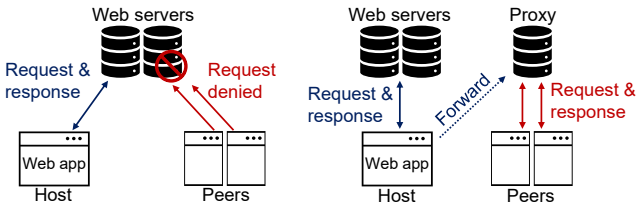
B. Hiding Personal Data

With the cross-device web technique, personal information can be protected from public view when web apps are used in public. For example, if a user logs on to Facebook during an online lecture presentation, that person's email addresses or mobile phone numbers are exposed to the audience by default (C2.1). In such a case, if the input and GUI elements related to the login could be passed to the user's personal smartphone, the private information would be hidden from the public view. Similarly, when a user wishes to share a specific photo on Google Photos on a public screen, other photos might accidentally be revealed (C2.2). This privacy problem can be solved by hiding personal photos on a private device and selectively displaying the intended photos on a public screen.

C. Enhancing User Interaction

The cross-device web technique resolves the inconvenience caused by the constrained input interface of a single device, providing convenient and enhanced user interactions. For example, take the scenario where a user is taking notes using Google Keep on a laptop during lectures or meetings (C3.1). When the user needs to draw a picture or take handwritten notes on the laptop, the mouse or trackpad input is probably not an optimal interface for the task. If the GUI elements of Google Keep can be passed to another device equipped with an input interface better suited for handwriting, such as a tablet computer with a stylus, the user will be provided with various input options for enhanced interactions with the application at hand. For another example, consider watching a YouTube video on a smart TV; it is difficult to finely handle the mouse pointer on the TV using a remote control to control the playback or search for

[1] We use the Cx.y notation to refer to each scenario in Sections II and VII.



(a) Accessing original web servers (b) Exploiting a proxy server

Fig. 2. Peer–server configurations.

videos (C3.2). If a smart TV could pass the YouTube control bar to a smartphone, then the user could easily handle the video on the TV with the smartphone's handy touch interface.

D. Supporting Collaborative Work

The functionality of web apps can be extended to support collaborative work by sharing the selected GUIs of a web app with other people's mobile devices. Suppose that a user is giving a presentation using Google Presentation with a laptop connected to a projector in a conference room (C4.1). If the presentation slides could be shared on the participants' devices, the slides would be directly accessed from the devices without the participants having to prepare hard copies. In addition, assume a scenario in which a user is working at home and coding through CodePen (C4.2). If the code-writing GUI and GUI displaying the results were shared with other coworkers in the workplace, the reviewing and revising process of the code development would be efficiently executed in a group.

III. Vulture Overview

Motivated by Section II, we propose Vulture, a system for cross-device web experiences without requiring modifications of web apps or browsers. We discuss the challenges and describe the architecture of our solution.

A. Challenges

Challenge 1. Orchestrating Peer–Server Configuration. To support the fine-grained GUI distribution of web apps, peers should be able to acquire web resources of various kinds, such as HTML documents, CSS documents, images, and fonts. One technical challenge is designing an efficient and effective peer–server configuration for resource distribution. There are two possible ways for peers to obtain the resources: (1) sending HTTP requests directly to original web servers and (2) exploiting proxy servers. Fig. 2(a) illustrates a naïve scheme in which peers directly send HTTP requests to original servers. A *host* is the primary device where a user selects the GUI elements, and *peers* are the secondary devices displaying the distributed GUI elements. This approach is straightforward and easy to implement but poses an access right issue. For example, many web apps identify users or clients using various methods, such as HTTP sessions [10], HTTP cookies [11], and tokens [12]. If a peer does not have access rights to a server, the peer cannot retrieve the web resources. Another approach for distributing web resources from hosts to peers is to exploit proxy servers, as illustrated in Fig. 2(b). While a host forwards web resources to a proxy server, peers fetch the resources from the proxy instead of the original server. Peers only need access rights to the proxy server. However, this approach has several problems. The proxy servers may encounter privacy and security problems because the private data of users are located outside of the users' devices.

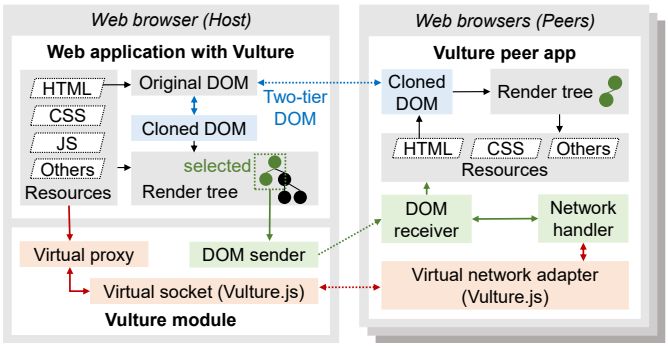


Fig. 3. The Vulture architecture.

Also, cost and maintenance issues exist because the proxy servers should accommodate all the traffic caused by users. To summarize, both approaches have inherent problems, and a new approach is required to handle the efficient distribution of web resources to peers.

Challenge 2. Ensuring Functional Consistency. Web apps operate based on interactions between JavaScript runtimes and browser engines. JavaScript codes, which define the logic of web apps, run in JavaScript runtimes. Browser engines handle the various functionalities provided by browsers, including rendering GUIs, networking, and receiving user inputs. In particular, browser engines provide DOM API dealing with GUIs for web apps. With the API, JavaScript runtimes detect user inputs and changes in view states, handle the inputs, and update view states. DOM-based GUI management operates under the assumption that web apps run in single-browser environments. In cross-device environments, the operation of DOM-based GUI management cannot be guaranteed. Thus, the GUI-distributed web apps themselves should handle user inputs from multiple devices and synchronize view states across the devices. The view state in a web app continuously changes at runtime; subsequently, the GUIs should correctly reflect the update on the display even when the GUI elements are transferred to another device. Furthermore, a web app sometimes involves changes in the URL when switching to the new view states [13], leading to reloading web resources and initializing JavaScript codes. Also, the user's input leads to executing the web app's JavaScript, which changes the code flow, the values of the data structures, and the internal state of the web app. Thus, the cross-device technique should properly maintain the internal states of the web apps and servers by synchronizing the user input between the devices.

B. Vulture Overview

The overall architecture of Vulture is illustrated in Fig. 3. The architecture is designed to support multiple peers, enabling a one-to-many distribution of GUI elements, but for brevity, we describe Vulture based on a single-peer configuration. The workflow of Vulture consists of two phases: the GUI distribution phase and the usage phase. In the GUI distribution phase, a user selects GUI elements on the host to distribute to the peer, and the DOM sender transmits the selected DOM to the peer. Transmitting the web resources to the peer is not trivial because the peer–server configuration should be well orchestrated. Vulture addresses this challenge by introducing the in-browser virtual proxy and a relevant API for virtual proxy, which are discussed in Section IV. During the usage phase,

where a user uses the GUI-distributed web app, the functional consistency of the web app should be ensured. Vulture addresses this challenge by providing the single-browser illusion with the two-tier DOM architecture discussed in Section V.

Vulture's actual GUI distribution is based on the in-browser virtual proxy and the two-tier DOM architecture. When a web app is loaded on the host, Vulture forwards the web resources of the web app to the virtual proxy server *a priori* and generates a cloned DOM on the host. When a user selects GUI elements to be distributed, the DOM sender converts the selected part of the cloned DOM into an HTML document, removes all the script tags from the document for the single-browser illusion, and sends the document to the peer. Upon receiving the document, the DOM receiver on the peer reconstructs the cloned DOM. Then, the peer parses the HTML document containing various URLs for web resources. During the parsing, the network handler blocks the network requests and sends the requests to the virtual proxy on the host. The virtual proxy sends the web resources to the peer by responding to the requests.

IV. In-Browser Virtual Proxy

We propose an in-browser virtual proxy scheme to solve the peer–server configuration issue discussed in Section III.A.

A. Necessity and Problem

Proxy servers—or remote proxies—are typically located on server computers or cloud servers outside of users' devices. We define an in-browser virtual proxy as a proxy server running *inside* users' browsers, hence technically bringing remote proxies into the browsers. This scheme resolves the access rights problem of peers, overcoming the privacy and cost issues of remote proxies. Since the virtual proxy runs in web browsers, private data are never exposed; thus, there is no need to maintain external servers to run remote proxies. In addition, the virtual proxy-based configuration has a performance advantage in terms of the round-trip time (RTT). Although an in-browser virtual proxy running in web browsers has many practical advantages, its development is not trivial. Unfortunately, web browsers cannot run server programs written in JavaScript because browsers do not provide the relevant APIs, such as the HTTP socket, which is an essential functionality for servers. Several attempts have been made to provide HTTP socket functionality—without modifying web browsers—to web apps, but they still lack support for the desired cross-device web experiences. Browsix [14] enables web browsers to run Node.js [15] server programs by developing a JavaScript-only operating system (OS) running in web browsers. The HTTP socket provided by the Browsix OS is limited to a single browser tab; that is, it is not possible to send HTTP requests to browsers on other devices. WebContainer [16] supports HTTP communications between different browser tabs, but the tabs must be in the same browser. To deliver web resources to other devices, the in-browser virtual proxy should be able to receive HTTP requests from other devices and send responses back to them.

B. Cross-Device Virtual HTTP

We propose a cross-device virtual HTTP scheme to solve the technical problem of developing an in-browser virtual proxy; this scheme allows web browsers on different devices to exchange HTTP requests and responses. The design of the cross-device virtual HTTP addresses two issues: (1) how to send HTTP messages to other devices with no HTTP sockets and (2) how to identify destinations where the messages should go.

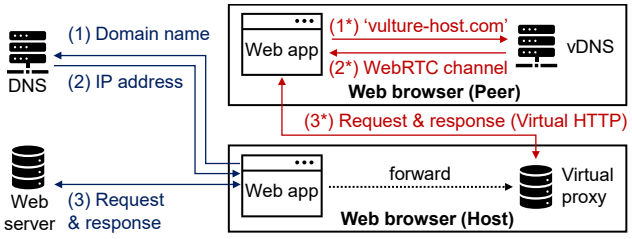


Fig. 4. The in-browser virtual proxy with the cross-device virtual HTTP.

The cross-device virtual HTTP cannot directly utilize the TCP/IP connections for HTTP communications because the sockets are not available in web apps. For the virtual HTTP connections, we employ WebRTC, which is practically the only way for web apps to communicate with each other. Before exchanging virtual HTTP requests and responses, devices first establish WebRTC connections. Virtual HTTP requests are then delivered via WebRTC data channels, and responses to the requests are sent via the same data channels. Meanwhile, domain names are used to identify the destinations of HTTP requests. The domain name system (DNS) translates domain names into Internet Protocol (IP) addresses. However, virtual HTTP cannot use domain names or IP addresses because the messages are sent via WebRTC. Instead, we introduce virtual domain names and a virtual DNS (vDNS) to identify those destinations under the virtual HTTP network. Virtual domain names point to specific devices in the same virtual HTTP network. The vDNS converts virtual domain names into designated WebRTC data channels. Fig. 4 illustrates the architecture of the in-browser virtual proxy and shows the flow of exchanging virtual HTTP requests and responses, here assuming that an HTML document has a *<img src='vulture-host.com/a.jpg'>* tag. When a web app sends a virtual HTTP request, the vDNS returns a WebRTC data channel regarding a virtual domain name. The web app then sends the request via the data channel.

C. Vulture.js: API for Virtual HTTP

To provide the cross-device virtual HTTP, we propose a new JavaScript API called Vulture.js, which provides API functions and syntax similar to Node.js programs. The main functionality of Vulture.js is to offer API functions for virtual HTTP sockets. The functions allow servers to receive virtual HTTP requests and send responses to the requests. Vulture.js also provides client-side API functions for web apps to use virtual HTTP and recognize destination servers when sending requests. TABLE I lists the API functions of Vulture.js. Fig. 5(a) shows a sample virtual server program written with the functions. The program opens a virtual HTTP socket with the createServer() and listen() functions. The listen() function designates the virtual domain name of the virtual server. To receive virtual HTTP requests, the server is connected to a virtual HTTP network with the connectToVirtualNetwork() function. Fig. 5(b) describes a sample web app code that sends a virtual HTTP request. When connecting to the virtual HTTP network, the web app saves the virtual domain name of the virtual server. All the HTTP requests to the virtual domain name are then sent to the virtual server. Vulture.js uses Service Worker API [17] to provide these functionalities, which intercept and control network requests.

TABLE I. API FUNCTIONS OF VULTURE.JS.

Function	Parameters	Return	Type
connectToVirtualNetwork()	WebRTC data channel, Virtual domain name	None	Static method
createServer()	Request listener	Virtual server	Static method
listen()	Virtual domain name	None	Virtual server's member method

```
let dc; // WebRTC data channel
let url_host = "vulture-host.com"; // virtual URL
Vulture.connectToVirtualNetwork(dc);
Vulture.createServer((req, res) => {
    // do something with req and res
}).listen(url_host);
```

(a) Server

```
let dc, url_host; // same as server
Vulture.connectToVirtualNetwork(dc, url_host);
fetch("http://" + url_host + "/style.css", …);
```

(b) Client

Fig. 5. Sample codes with the Vulture.js.

V. TWO-TIER DOM ARCHITECTURE

We propose a two-tier DOM architecture to handle the functional consistency issue discussed in Section III.A.

A. Two-Tier DOM with Cloning

The two-tier DOM architecture is designed to provide a single-browser illusion to GUI-distributed web apps. As described earlier, JavaScript runtime executes the core logic of web apps and uses DOM to handle GUIs. The two-tier DOM architecture aims to maintain the interactions between the JavaScript runtime and DOM during the cross-device web experience. The two-tier architecture clones the DOM of web apps using two types of DOMs—original DOM and cloned DOM. The architecture allows the JavaScript runtime in the host to interact with the original DOM, while distributed GUIs are handled with the cloned DOM in each peer. The original DOM is the same as the DOM running in a single browser. In this way, the JavaScript runtime on the host cannot tell whether the GUIs of the web app are distributed or not. Consequently, the two-tier DOM architecture executes all the core operations of web apps on the host, such as running JavaScript codes and handling user inputs. The peer updates the screen content by receiving the view state from the host (i.e., the view state reproduction). The peer also forwards all user inputs to the host (i.e., user input delegation). Sections V.B and V.C detail the view state reproduction and user input delegation, respectively.

B. View State Reproduction

The view state changes because of the execution of JavaScript or URL transitions. When the execution of JavaScript on the host changes the view state corresponding to the GUI elements transferred to the peer, the host detects the change and sends the changed DOM as an HTML segment to the peer. When a URL transition occurs and the view state is altered, the host checks whether the loaded page is in the same application context as the existing page. The criterion for this check is whether the GUI elements on the peer can be found in the DOM tree of the new page. From the peer's perspective, the URL transition is merely a change in the view state of the GUI elements. Subsequently, the peer obtains the changed DOM and updates the view state. Meanwhile, the media elements continuously draw frames on display without making changes in the DOM tree; thus, the media elements cannot be synchronized by tracking the view state. To resolve this issue, the host streams the media to the peer. The two-tier DOM architecture intercepts the stream on the host's web app, transmits the stream to the peer, and connects the stream to the media element in the peer.

C. User Input Delegation

The two-tier DOM architecture handles user inputs across devices by delegating all user inputs from the peer to the host. Because all user inputs occurring on the peer are delivered to the host and centrally processed there, the architecture effectively prevents discrepancies in the operation flow because of duplicated code execution. Note that the architecture synchronizes the processing of user inputs, even in multi-peer configurations, because all user inputs are delegated to the host. To address the issue of input heterogeneity, the host translates the input event received from the peer into an event that plays the same role on the host. Also, to deal with the situations where the input event contains the coordinates on the screen or the screen sizes of the host and the peer are different, we designed the "User Input Handler" to convert the coordinates from the peer to the corresponding location on the host's screen. With this technique, the two-tier DOM architecture provides a seamless experience in user input between heterogeneous devices.

VI. VULTURE IMPLEMENTATION

With the core techniques developed above, we implemented Vulture to provide cross-device experiences in real web environments. Also, we optimized the performance of Vulture.

A. Implementation

We implemented the Vulture prototype in real-world browser environments. The implementation supported various devices and platforms: specifically, a Windows-based laptop, an Android smart TV, an Android tablet computer, and an Android smartphone. Vulture prototype was developed as a Chrome extension [18]. In the case of Android devices, we used a modified version of Chrome, the Kiwi browser [19], because the Android version of Chrome does not support the extensions. On the host, the extension handles the host's functionalities while interacting with the web app. On the peer, the extension just opens up a new tab running the Vulture peer app, where the peer's functionalities work.

The various functionalities of the Vulture prototype were implemented with Chrome APIs [20] and Web APIs [21]. For example, we used the MutationObserver interface [22] to detect the change in view states. GUI element selection was implemented by referring to "Select an element" in Chrome DevTools [23]. When a user places the mouse cursor at a specific point, the relevant DOM node is highlighted in blue. By clicking it, the user selects the node. For mouse-less devices such as smartphones, the user could highlight a DOM node by touching a specific point and then select the node by dragging from the point. Note that these Web APIs and WebExtensions API [24] are being standardized; thus, the Vulture prototype could be readily integrated into non-Chrome-based browsers in due course.

TABLE II. WEB APPS FOR THE VULTURE TEST.

Case	Web app	Input*
C1.1	Twitch, YouTube Gaming, TED, Coursera, Delish	C
C1.2	OpenStreetMap, Waze, Bing Maps, Google Maps, Airbnb	D
C2.1	Facebook, Google, GitHub, Instagram, Yahoo	T
C2.2	Google Photos, Amazon Photos, Dropbox, OneDrive, Flickr	C
C3.1	Google Keep, Chrome Canvas, OneNote, Kleki, Sketch.IO	C, D
C3.2	YouTube, Vimeo, Dailymotion, Netflix, Facebook Watch	C, D
C4.1	Google Presentation, SlideShare, PowerPoint, SlideServe, Slides	C, D
C4.2	CodePen, Ideone, JSFiddle, Jupyter Notebook, W3Schools	C, D, T
Others	MDN Web Docs, Medium, Reddit, Google Scholar, Stack Overflow, Quora, Gmail, Outlook, Amazon, Walmart	C

* C denotes clicking/touching; T denotes typing; D denotes dragging/swiping.

B. Performance Optimization

Optimizing the performance of the resource transmission mechanism is important from a user experience point of view. Vulture employs Server Push [25] to accelerate resource transmission. Web servers send one response to one request, while Server Push allows web servers to send multiple responses for reduced latency and improved page load speed. Unfortunately, Server Push has barely been adopted in real web environments because the optimal operation of Server Push depends on many fine-tuned factors that are difficult to be decided *a priori* [26]. In the case of Vulture, the in-browser virtual proxy should know exactly what web resources should be delivered to peers. The DOM sender on the host finds all the URLs when generating the HTML document regarding the cloned DOM. Then, when transmitting the HTML document, the DOM sender transfers the web resources related to the URLs together. The DOM sender does not look for URLs in other web resources, such as CSS files, for optimized performance; in this case, the network handler in the peer deals with missing URLs.

VII. EVALUATION

We evaluated the cross-device functionality of Vulture in real environments. We also analyzed the performance characteristics of Vulture running in diverse hardware and network environments.

A. Experiment Setup

For the evaluation, we selected 50 web apps to cover the various characteristics of web apps. TABLE II lists the selected web apps. We chose five web apps for each of the eight scenarios discussed in Section II, including the eight web apps mentioned in Section II, totaling 40 web apps. We then selected additional 10 web apps that we considered to be common and widely used in the real world. When selecting these web apps, we also considered various factors affecting the performance of Vulture to make the evaluation closely represent the diverse characteristics of the web in real life. The experiments were conducted by following the usage scenarios listed in the first column of TABLE II. We configured the host and peer combinations with a laptop (Intel Core i5-10210U CPU) and a smartphone (Snapdragon 820 MSM8996 SoC AP). To consistently evaluate the system configuration, we experimented with the desktop version of a web app for the laptop and smartphone. We also controlled the network bandwidth to 20 Mbps by configuring the Wi-Fi router in the lab.

TABLE III. COMPARISON BETWEEN VULTURE AND EXISTING SOLUTIONS.

Case	AnyDesk	Chromecast	XDBrowser*	Vulture
C1.1		✓		✓
C1.2			✓	✓
C2.1				✓
C2.2			✓	✓
C3.1				✓
C3.2		✓		✓
C4.1			✓	✓
C4.2	✓		✓	✓

* XDBrowser works only when a user is not involved with a login session.

In the Vulture architecture, the in-browser virtual proxy plays an important role. However, the proxy can be located outside the hosts (i.e., remote proxy). To evaluate the performance advantage of the in-browser virtual proxy compared with the remote proxy, we built a remote proxy running on Amazon Web Services (AWS). The preliminary measurements showed that the latency between a device in the lab and various servers on the AWS ranged between 10 ms and more than 200 ms. Thus, we selected 100 ms latency as the latency for the remote proxy.

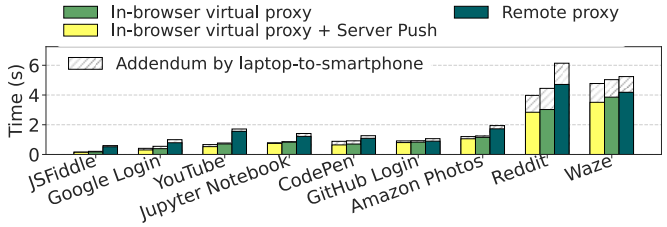
B. Functionality

We first conducted an analysis of the cross-device functionality of Vulture in comparison with the existing solutions. TABLE III summarizes the functionality of AnyDesk [3], Chromecast [5], XDBrowser [9], [27], and Vulture. The eight usage scenarios discussed in Section II were checked for each solution to see if they could support the GUI distribution. Our analysis shows that existing solutions were severely limited in providing fine-grained, cross-device experiences on a range of real applications. Apart from simple mirroring (C4.2), AnyDesk virtually provided no GUI distribution. Chromecast only supported C1.1 and C3.2, where the video component of the application was transmitted from mobile to big-screen devices. We found that XDBrowser provided more diverse cross-device functionalities (C1.2, C2.2, C4.1, and C4.2) compared to AnyDesk and Chromecast, but its practical use was limited due to the inability to migrate media elements and lack of consideration for authentication issues. XDBrowser supported the usage scenarios only when a user was not involved with server-side logins.

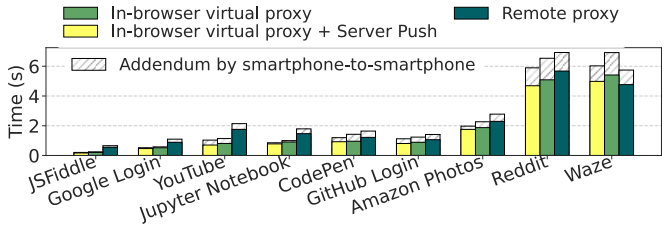
After verifying Vulture's functional superiority, we evaluated its applicability using the 50 web apps listed in TABLE II. To simplify the experiments, we set the laptop and the smartphone as the host and peer, respectively. We manually selected the most appropriate GUI elements to distribute to the peer based on each web app's usage scenario. We tested whether we could distribute the GUI of web apps at a fine-grained level using Vulture. We also checked the functional correctness of the web apps after the GUI distribution. The results confirmed that Vulture provided fine-grained GUI distribution and maintained functional consistency for all the tested web apps.

C. Performance

We evaluated the performance aspect of Vulture to validate its usefulness and effectiveness in real cross-device web environments. We specifically employed the RAIL model [28], which is known to provide a user-centric performance model for web apps. The RAIL model categorizes the performance of web



(a) Laptop-to-laptop and laptop-to-smartphone



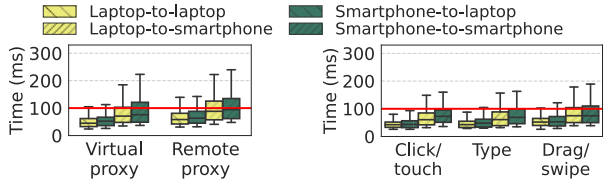
(b) Smartphone-to-laptop and smartphone-to-smartphone

Fig. 6. Resource transmission time.

apps into four domains: response, animation, idle, and load. Among these domains, response time and load time were highly related to the performance of Vulture. The transmission process for the GUI elements corresponded to the load time of the peer devices. Meanwhile, the processes for view state reproduction and user input delegation are relevant to the response time from the viewpoint of the peer and host, respectively.

GUI and resource transmission time. According to the load time of the RAIL model, the user experience improves with the fast migration of the GUI elements and related resources to the peer. When a user selects a GUI element on the host to send the element to the peer, the overall process falls into two phases: (1) sending a base HTML document about the cloned DOM and (2) sending web resources. In general, sending web resources requires more traffic than sending the DOM. Sending the DOM does not involve proxy servers, and there is only one data transmission: the host sends a base HTML document directly to the peers via the WebRTC data channel. This phase depends on the performance of WebRTC. On the other hand, sending web resources is affected by several factors—especially peer–server configuration and Server Push. For the evaluation, we measured the amount of web resources required by the GUI distribution for each web app listed in TABLE II and selected 10 specific web apps based on the measurement: low (bottom one-third), mid (between one-third and two-thirds), and high (top one-third). We selected JSFiddle (0.04 MB), Google Login (0.1 MB), and YouTube (0.2 MB) in the low group; Jupyter Notebook (0.4 MB), CodePen (0.5 MB), and GitHub Login (0.7 MB) in the mid group; and Amazon Photos (1.7 MB), Reddit (3.8 MB), and Waze (7 MB) in the high group.

Fig. 6 summarizes the performance of Vulture with respect to the web resource transmission time. Here, the load time was measured as a time delay between when the peer started parsing the base HTML document and when the load event [29] was fired on the peer, and we measured the load time ten times for each web app. The median load time for each chosen app is shown in the figure. Fig. 6(a) and (b) illustrate the load time of the peer for the laptop-host and smartphone-host cases, respectively. The hatched areas in the figures indicate the additional delay when the peer was configured as a smartphone,



(a) View reproduction time (b) User input delegation time

Fig. 7. Synchronization performance.

compared to the laptop-peer cases. The results suggested two important facts. First, the in-browser virtual proxy reduced the average transmission time by 25.23% compared to the remote proxy. This result arose from the fact that the in-browser virtual proxy had the advantage of low latency over the remote proxy. However, in cases where the host was a smartphone, the virtual proxy could make a slower transmission than the remote proxy, as observed in the Waze case. This was attributed to the weaker computing power of the smartphone and the large-sized web resources, which mitigated the in-browser virtual proxy's low latency characteristic. Nevertheless, the in-browser virtual proxy performed better for almost all cases than the remote proxy, making it acceptable to use smartphones as the host. Second, Server Push effectively reduced the transmission time by an average of 10.58%. Thus, the in-browser virtual proxy with Server Push shortened the mean transmission time by 38.47% compared to the remote proxy. The reduction in transmission time was attributed to the decreased number of HTTP requests facilitated by Server Push.

View reproduction time. An immediate reflection of the view changes from the host to peers improves the response time of the RAIL model, subsequently leading to a good user experience. Note that according to the RAIL model, the criterion for users to feel the response is immediate is a 100 ms delay. To evaluate the responsiveness of view changes on Vulture, we measured the response time of view changes, i.e., the view state reproduction time. For the web apps listed in TABLE II, we executed the given scenarios for 10 seconds and measured the time to transmit the view changes from the host to the peer. The time was measured with various host-to-peer combinations and also with two types of proxy—the in-browser virtual proxy and the remote proxy. The in-browser virtual proxy was optimized with Server Push.

Fig. 7(a) shows the distributions of the view state reproduction time. For the in-browser virtual proxy, the response time was within 100 ms for almost all cases, indicating that the view state reproduction could be perceived by users as almost *immediate*. We observed that the in-browser virtual proxy reduced the average view reproduction time by 20.46%. Note that view changes often accompany additional resource transmissions; thus, the result suggested that the optimized resource transmission could improve the efficiency of the view reproduction. Further explaining the results, Fig. 7(a) shows that the computing power of the peer device was more critical than that of the host device. Users using the smartphone peer could feel that the response was not immediate because the reproduction time sometimes exceeded 100 ms, especially with the remote proxy.

User input delegation time. The fast delegation of the user input from a peer to the host results in a good user experience related to the response time of the RAIL model. Three types of user input—clicking or touching, typing, and dragging or

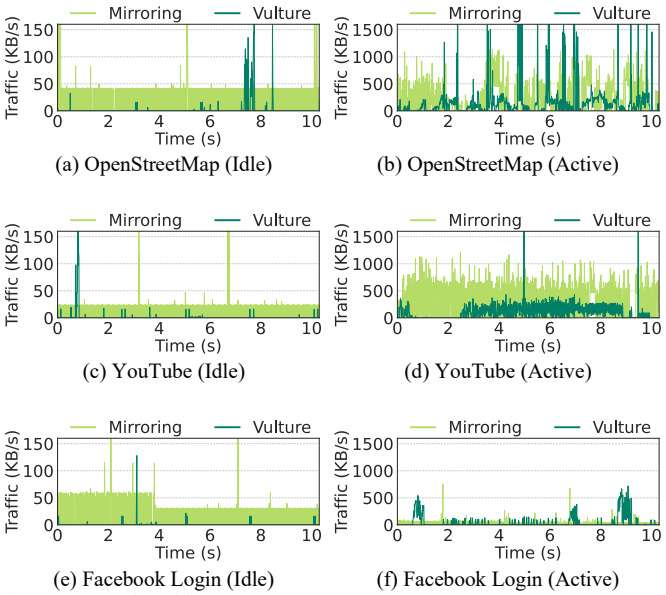


Fig. 8. Network traffic traces.

swiping—are generally used for web apps running on diverse devices. Clicking, touching and typing consist of only two events: the start and the end of the input. Dragging and swiping comprises multiple events, including the start and end points, because an event is raised whenever a mouse pointer or finger moves over pixels. The number of events raised is specific to the types of user inputs and certainly affects the overall response time. We investigated which types of user input were primarily used on the peers for each scenario. The "Input" column of TABLE II summarizes the results. For the evaluation, we chose four web apps for each type of user input to analyze the implications of the input delegation time for the response time.

Fig. 7(b) shows the distributions of the user input delegation time, that is, the response time taken to transfer user inputs from the peer to the host for each input type. Overall, the user input delegation time was considered *immediate* because almost all user inputs were delegated from the peer to the host within 100 ms. The figure also suggests that the sort of user input barely affected the response time, even for the dragging or swiping. Specifically, we measured the user input delegation time with different device combinations. Similar to the view state reproduction, Fig. 7(b) shows that the computing power of the peer had more of an impact on the user input delegation time than that of the host.

D. Network Traffic

In a mobile web environment where network access is costly, network traffic should be optimized [30], [31]. We evaluated the network efficiency of Vulture by measuring the network traffic generated at runtime. For the comparative analysis, we implemented the WebRTC-based screen-sharing scheme [32], which is considered the baseline approach. We selected three representative web apps for evaluation based on the premeasured network traffic. The three apps—OpenStreetMap, YouTube, and Facebook Login—represent the cases for large, medium, and small amounts of traffic, respectively.

Fig. 8 shows that Vulture generated less traffic than the mirroring approach overall. In the figure, idle and active represent the static and dynamically changing view states of a web app, respectively. Fig. 8(a), (c), and (e) show that, during the idle state, Vulture generated significantly less network traffic than the mirroring approach. This is because the network traffic of Vulture was generated only when the view state of the host changed, while the mirroring approach continuously streamed the entire screen image of the host. Fig. 8(b), (d), and (f) show the network traffic for the active state. Vulture and the mirroring approach generated high network traffic in the active state, but Vulture traffic was far less than that of the mirroring approach, indicating the efficiency of the synchronization process of Vulture. In short, the evaluation showed that Vulture performed efficiently in terms of the network traffic required for cross-device web functionality.

E. User Study

To assess the cross-device usability of Vulture, we performed an institutional review board (IRB)-approved user study with 18 participants. The group consisted of 14 males and 4 females, with 14 in their twenties and 4 in their thirties. The participants were recruited through bulletin boards at school and online school communities. Each participant was provided with three different devices—a smartphone, a tablet computer, and a laptop computer. We explained the purpose and key features of Vulture to the participants and then conducted the user study in two steps. First, the participants were asked to use their own cross-device web experiences by selecting five web apps that they thought would be useful with distributed GUI elements across multiple devices. Second, the participants experienced the eight cross-device usage scenarios described in Section II. For each scenario, we used a 5-point Likert scale to inquire about the usability of the cross-device experiences provided by Vulture in comparison to single-device web experiences.

In the first step, the participants created various use cases beyond the eight scenarios presented in the paper. This was indeed possible because Vulture provided fine-grained GUI distribution. Interestingly, many participants wished to improve the visibility of web pages containing too much content in a single viewport. One specific example was the use of online course platforms, such as Coursera. The Coursera page on a laptop computer consists of three key components—a video, subtitles, and notes. The participants watched a video on full screen with a laptop and read subtitles or notes on a tablet or smartphone to improve the user experience. Another interesting use of Vulture was to enhance user interaction. When editing a photo on Pixlr with a mouse on a laptop, the user first transferred the image canvas to a tablet and used a stylus pen to draw more conveniently. The toolbox component of the application was also transferred to a smartphone for better usability. Aside from these specific cases, the participants suggested diverse use cases, indicating that Vulture is not limited to specific web apps but can be useful and practical in many applications.

Fig. 9 shows the result of the second step. Overall, the participants were very positive about cross-device experiences. Note that *Cx.y* and *others* in the figure stand for each scenario in Section II and for the user-chosen scenarios, respectively. The number in parentheses is the average score. Among the four categories in Section II, "hiding personal data" showed the highest average score (4.53). Specifically, C2.1 delivered the most positive feedback (average score of 4.56). Some participants commented that features like C2.1 were impressive

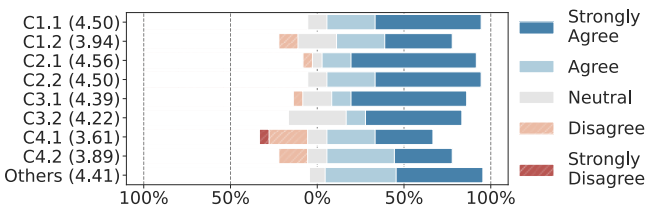


Fig. 9. User study results on the usability of Vulture.

and useful because they often had uncomfortable experiences of providing personal information on a public screen. On the other hand, C4.1 showed the lowest average score (3.61). Some participants may have already experienced similar cross-device functionality with other existing applications, resulting in a less impressive experience with Vulture in this category.

VIII. Related Work

Efforts have been made to utilize multiple devices simultaneously for improved user experience by distributing a GUI [1], [33]. The traditional approach for a single user to use multiple devices at one time is called mirroring, which simply displays the screen of an app running on one device on another. Several commercial products exist for mirroring solutions [2]–[5], [34], [35]. The mirroring approach is generally applicable to various kinds of apps without having to consider synchronization among devices; the app runs on one device, and other devices simply display the shared screen. However, with the mirroring scheme, the GUI elements of an app cannot be pinpointed for selective transmission.

More advanced approaches have been introduced to distribute the GUI elements of native apps across multiple devices. SAMD [36] and MSA [37] proposed frameworks for creating new apps that provide a cross-device experience. XDSession [38] and Husmann et al. [39] proposed tools for testing or debugging cross-device apps. These frameworks or tools still require effort from developers; subsequently, several works, such as CollaDroid [40] and UIWear [41], proposed schemes that help developers easily convert existing Android apps into cross-device versions. The development of a new cross-device app benefits from this approach, but existing apps must be reauthored with effort. Meanwhile, FLUID [6], FLUID-XP [7], and PRUID [8] can convert existing apps into cross-device versions without reauthoring efforts by transmitting the code or the data required for an app to operate on a peer device. These solutions are applicable only to Android-based apps; thus, they are not generally applicable to heterogeneous devices or computing platforms. To summarize, native app-based solutions are limited to specific platforms, while Vulture provides cross-device functionalities across heterogeneous devices.

Research has also been conducted to provide a cross-device experience for web apps that inherently run in heterogeneous environments. Several approaches have been proposed for the solution. First, similar to the mirroring method for native apps, the screens of web apps have been simply shared with other devices. Screensharing with WebRTC [32] and Chromecast [5] belongs to this category. Second, various tools or frameworks have been proposed to support the development of cross-device web apps. The tools include Ghiani et al. [42], Virtual Browser [43], XDStudio [44], Tandem Browsing Toolkit [45], Liquid.js [46], Panelrama [47], and Crow API [48]. Although these tools help develop cross-device web apps, developer efforts are still required. Finally, efforts have been made to provide cross-device web experiences without reauthoring existing apps. XDBrowser [9], [27] belongs to this category. XDBrowser primarily focuses on the HCI issues of a cross-device web app; thus, the solution is hardly applicable to real web apps. Specifically, XDBrowser lacks solutions for the essential challenges in providing cross-device web experiences in real life, such as authorization, the synchronization of GUI elements, and the handling of user input. Vulture addresses and solves these technical issues to provide cross-device functionalities in the wild.

IX. Novelty of the Proposed Work

No modifications of applications. Application reauthoring hampers the widespread use of cross-device web experiences. First, reauthoring shifts the development burden onto developers. Although cross-device experiences are useful, developers might not want to put their efforts into developing such functionality. This is the primary reason most applications currently do not provide cross-device GUI distribution. Second, developers might not foresee all use cases of cross-device experiences; that is, it is not pragmatic for developers to cover various usage scenarios. With Vulture, users can select and distribute GUI elements the way they want to use the applications on their devices; thus, the developers are freed from the diversity issue in implementation.

Application-level approach. Previous research on fine-grained GUI distribution, such as FLUID, FLUID-XP, and PRUID, adopted an OS-level approach, where the underlying operating systems are modified for the solution. The solutions are highly dependent on the operating systems, such as Android. On the contrary, Vulture solves the platform dependency issue by exploiting the meta-platform characteristic of web applications. Modifying the browser is a possible solution for cross-device web experiences, but it can lead to the browser dependency issue, requiring a specific browser for users. With the application-level approach, Vulture is compatible with the standard Web APIs and thus works consistently on various platforms and browsers.

X. Conclusion

We presented key challenges and solutions that should be addressed to provide readily usable cross-device functionalities in real-world web environments. We proposed an in-browser virtual proxy to efficiently distribute web resources from the host to peers. To this end, we developed a cross-device virtual HTTP and an API for the scheme. For the functional consistency among distributed web GUIs, we proposed a two-tier DOM architecture that provides a single-browser illusion to web apps. We hope that Vulture can help research in the relevant fields and provide directions in web standards development for supporting fully functional cross-device web experiences.

Acknowledgments

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2018-0-00532, Development of High-Assurance (≥EAL6) Secure Microkernel).

References

[1] F. Brudy et al., "Cross-Device Taxonomy: Survey, Opportunities and Challenges of Interactions Spanning Across Multiple Devices," in *Proc. 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*, 2019, pp. 1–28.

[2] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper, "Virtual network computing," *IEEE Internet Comput.*, vol. 2, no. 1, pp. 33–38, 1998.

[3] "AnyDesk," https://anydesk.com/en

[4] "Chrome remote desktop," https://remotedesktop.google.com/access/

[5] "Chromecast," https://store.google.com/product/chromecast

[6] S. Oh et al., "FLUID: Flexible User Interface Distribution for Ubiquitous Multi-Device Interaction," in *Proc. 25th Annual International Conference on Mobile Computing and Networking (MobiCom '19)*, 2019, pp. 1–16.

[7] S. Lee et al., "FLUID-XP: Flexible User Interface Distribution for Cross-Platform Experience," in *Proc. the 27th Annual International Conference on Mobile Computing and Networking (MobiCom '21)*, 2021, pp. 762–774.

[8] M. Cui et al., "PRUID: Practical User Interface Distribution for Multi-surface Computing," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 679–684.

[9] M. Nebeling, "XDBrowser 2.0: Semi-Automatic Generation of Cross-Device Interfaces," in *Proc. 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*, 2017, pp. 4574–4584.

[10] "A typical HTTP session - HTTP | MDN," https://developer.mozilla.org/en-US/docs/Web/HTTP/Session

[11] "Using HTTP cookies - HTTP | MDN," https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

[12] "JSON Web Token Introduction - jwt.io," https://jwt.io/introduction/

[13] "Single-page application vs. multiple-page application," https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58

[14] B. Powers, J. Vilk, and E. D. Berger, "Browsix: Bridging the gap between unix and the browser," in *Proc. Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, 2017, pp. 253–266.

[15] "Node.js," https://nodejs.org/en/

[16] "WebContainer," https://github.com/stackblitz/webcontainer-core

[17] "Service Worker API - Web APIs | MDN," https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

[18] "What are extensions? - Google Chrome," https://developer.chrome.com/extensions

[19] "Kiwi Browser," https://play.google.com/store/apps/details?id=com.kiwibrowser.browser

[20] "Chrome APIs - Google Chrome," https://developer.chrome.com/extensions/api_index

[21] "Web APIs - Google Chrome," https://developer.chrome.com/apps/api_other

[22] "MutationObserver - Web APIs | MDN," https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver

[23] "CSS features reference - Chrome Developers," https://developer.chrome.com/docs/devtools/css/reference/#select

[24] "Browser Extensions - Mozilla | MDN," https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions

[25] "Server Push," https://web.dev/performance-http2/#server-push

[26] N. Kansal, M. Ramanujam, and R. Netravali, "Alohamora: Reviving {HTTP/2} Push and Preload by Adapting Policies On the Fly," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, 2021, pp. 269–287.

[27] M. Nebeling and A. K. Dey, "XDBrowser: User-Defined Cross-Device Web Page Designs," in *Proc. 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*, 2016, pp. 5494–5505.

[28] "Measure performance with the RAIL model," https://web.dev/rail/

[29] "Window: load event - Web APIs | MDN," https://developer.mozilla.org/en-US/docs/Web/API/Window/load_event

[30] V. Agababov et al., "Flywheel: Googles Data Compression Proxy for the Mobile Web," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, 2015, pp. 367–380.

[31] R. Netravali and J. Mickens, "Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, 2018, pp. 249–266.

[32] "Screensharing with WebRTC | Google Developers," https://developers.google.com/web/updates/2012/12/Screensharing-with-WebRTC

[33] F. Paternò, "Concepts and design space for a better understanding of multi-device user interfaces," *Univers. Access Inf. Soc.*, vol. 19, no. 2, pp. 409–432, 2020.

[34] "TeamViewer," https://www.teamviewer.com/en-us/

[35] "Vysor," https://www.vysor.io/

[36] J. Lee, H. Lee, B. Seo, Y. C. Lee, H. Han, and S. Kang, "SAMD: Fine-Grained Application Sharing for Mobile Collaboration," in *2018 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2018, pp. 1–10.

[37] Z. Chen, T. Wang, J. Xue, and Z. Shao, "MSA: A Novel App Development Framework for Transparent Multiscreen Support on Android Apps," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 42, no. 10, pp. 3171–3184, 2023.

[38] M. Nebeling, M. Husmann, C. Zimmerli, G. Valente, and M. C. Norrie, "XDSession: Integrated Development and Testing of Cross-Device Applications," in *Proc. 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '15)*, 2015, pp. 22–27.

[39] M. Husmann, N. Heyder, and M. C. Norrie, "Is a Framework Enough? Cross-Device Testing and Debugging," in Proc. *8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '16)*, 2016, pp. 251–262.

[40] J. Zheng et al., "CollaDroid: Automatic Augmentation of Android Application with Lightweight Interactive Collaboration," in *Proc. 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW '17)*, 2017, pp. 2462–2474.

[41] J. Xu, Q. Cao, A. Prakash, A. Balasubramanian, and D. E. Porter, "UIWear: Easily Adapting User Interfaces for Wearable Devices," in *Proc. 23rd Annual International Conference on Mobile Computing and Networking (MobiCom '17)*, 2017, pp. 369–382.

[42] G. Ghiani, F. Paternò, and C. Santoro, "Push and Pull of Web User Interfaces in Multi-Device Environments," in *Proc. International Working Conference on Advanced Visual Interfaces (AVI '12)*, 2012, pp. 10–17.

[43] B. Cheng, "Virtual Browser for Enabling Multi-device Web Applications," in *Proc. Workshop on Multi-device App Middleware (Multi-Device '12)*, 2012.

[44] M. Nebeling, T. Mintsi, M. Husmann, and M. Norrie, "Interactive Development of Cross-Device User Interfaces," in *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*, 2014, pp. 2793–2802.

[45] T. Heikkinen, J. Goncalves, V. Kostakos, I. Elhart, and T. Ojala, "Tandem Browsing Toolkit: Distributed Multi-Display Interfaces with Web Technologies," in *Proc. International Symposium on Pervasive Displays (PerDis '14)*, 2014, pp. 142–147.

[46] A. Gallidabino and C. Pautasso, "The Liquid User Experience API," in *WWW '18 Companion: The 2018 Web Conference Companion*, 2018, pp. 767–774.

[47] J. Yang and D. Wigdor, "Panelrama: Enabling Easy Specification of Cross-Device Web Applications," in *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*, 2014, pp. 2783–2792.

[48] S. Park, J. Lee, and H. Cha, "Crow API: Cross-device I/O Sharing in Web Applications," in *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, 2023, pp. 1–10.