Crow API: Cross-device I/O Sharing in Web Applications

Seonghoon Park, Jeho Lee, and Hojung Cha[*]
Department of Computer Science
Yonsei University
Seoul, Republic of Korea
{park.s, jeholee, hjcha}@yonsei.ac.kr

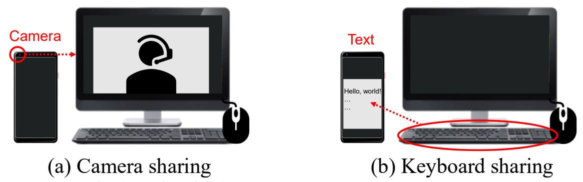
***Abstract*—Although cross-device input/output (I/O) sharing is useful for users who own multiple computing devices, previous solutions had a platform-dependency problem. The meta-platform characteristics of web applications could provide a viable solution. In this paper, we propose the Crow application programming interface (API) that allows web applications to access other devices' I/O through standard web APIs *without* modifying operating systems or browsers. The provision of cross-device I/O should resolve two key challenges. First, the web environment lacks support for device discovery when making a device-to-device connection. This requires a significant effort for developers to implement and maintain signaling servers. To address this challenge, we propose a serverless Crow connectivity mechanism using devices' I/O-specific communication schemes. Second, JavaScript runtimes have limitations in supporting cross-device inter-process communication (IPC). To solve the problem, we propose a web IPC scheme, called Crow IPC, which introduces a proxy interface that relays the cross-device IPC connection. Crow IPC also provides a mechanism for ensuring functional consistency. We implemented the Crow API as a JavaScript library with which developers can easily develop their applications. An extensive evaluation showed that the Crow API provides cross-device I/O sharing functionality effectively and efficiently on various web applications and platforms.**

***Index Terms*—cross-device I/O sharing, web application, mobile web, WebRTC, application programming interface (API).**

I. Introduction

With the widespread use of diverse mobile devices, users commonly own multiple computing devices, ranging from traditional desktop computers to wearable devices. Easy access to multiple devices has brought attention to cross-device input/output (I/O) sharing techniques, which allow user applications to utilize other devices' I/O whose functionality was originally developed for single-device environments. These techniques offer useful experiences in two ways. First, I/O sharing is helpful when a device does not have an appropriate I/O for an application. For example, suppose a user is running a video-conferencing application on a desktop computer with no camera. With cross-device I/O sharing, the user can access his or her smartphone camera instead, as shown in Fig. 1(a). Second, I/O sharing allows users to access more convenient I/O options. Assume that a user is using a virtual keyboard to edit texts on a smartphone. Using the virtual keyboard may degrade the user experience because the keyboard takes up certain areas of the screen. As illustrated in Fig. 1(b), the user would utilize the entire screen if he or she used a desktop's physical keyboard.

Cross-device I/O sharing can be provided via native OS support or via application-level protocols. Previous solutions mostly focused on the native OS support [1]–[4]. However, OS modifications are difficult and would hurt the portability of apps that rely on a particular OS's functionality. Application-level approaches shift the development burden to application



(a) Camera sharing (b) Keyboard sharing

Fig. 1. Cross-device I/O sharing.

programmers, forcing them to translate the underlying platform idiosyncrasies into platform-neutral abstractions. In this regard, web applications can help the programmers alleviate the burden due to the "meta-platform" characteristic. Modern web browsers provide platform-neutral abstractions of I/O devices, such as cameras, microphones, and sensors, with the standard web APIs [5]. In addition, current browsers support progressive web applications (PWAs) [6], which provide native application-like user experiences. The meta-platform characteristic of web applications offers great opportunities for cross-device I/O sharing on heterogeneous devices.

Despite this potential, cross-device I/O sharing in web applications has not been widely adopted because the approach requires considerable efforts in reauthoring applications to provide such functionalities. Thus, a new API mitigating the efforts would be greatly helpful. Two practical challenges exist in designing such an API. First, the web environment lacks support for device discovery when making a device-to-device or peer-to-peer connection. Web Real-Time Communication (WebRTC) [7] is practically the only way to support communications in web applications, but signaling servers are explicitly required for the operations. Second, the inherent characteristics of JavaScript runtime make it difficult for web applications to use other devices' I/O. Cross-device I/O sharing requires inter-process communication (IPC) functionality between devices. However, JavaScript runtimes provide no direct IPC to external processes.

In this paper, we propose an API, called the Crow API, to help web developers provide cross-device I/O sharing in web applications. By simply adding a few lines of Crow API code to the application, web developers can easily provide I/O sharing functionality in their applications. For practical and efficient device discovery and connectivity, we propose the Crow connectivity mechanism, which allows web applications to establish WebRTC connections without a signaling server. The mechanism basically exploits quick response (QR) codes and MousePath [8] to establish connections covering various types of computing devices. To address JavaScript's inherent limitations for providing cross-device I/O, the API provides a cross-device IPC mechanism, called the Crow IPC. The new IPC enables JavaScript runtimes to communicate with browser processes on other devices and synchronizes I/O data between devices.

[*] Corresponding author

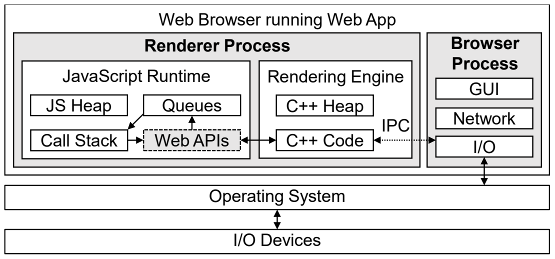


Fig. 2. Web workflow from the I/O perspective.

We implemented the Crow API as a JavaScript library so that web developers can readily import it into their applications with minimum effort. Additionally, we developed a browser extension that automatically inserts the Crow API into web applications. With this feature, cross-device I/O sharing functionality can be implemented in existing web applications.

II. Background

We describe the workflow of web applications and web APIs related to I/Os. Then, we discuss several challenges in supporting cross-device I/O sharing in web applications.

A. WebRTC

WebRTC is the standard and only way to provide peer-to-peer communications in web applications. WebRTC especially aims to provide audio and video communication. To make WebRTC connections between two peers, the peers exchange their session description protocols (SDPs), which include diverse information needed to establish WebRTC connections. The SDP exchange is bidirectional—a peer (caller) sends an SDP offer to another peer (callee), then the caller receives an SDP answer from the callee. The WebRTC API does not include peer discovery or device discovery mechanisms; thus, signaling servers are necessary to discover other devices and exchange SDPs with the devices before the WebRTC connection is established.

B. Web APIs for I/O Devices

The overall flow of a web application running on a modern web browser is illustrated in Fig. 2. The single-threaded *renderer process* is built up with a JavaScript runtime and a rendering engine. JavaScript codes are interpreted by a browser and executed in a JavaScript runtime. The JavaScript runtime cannot make a direct IPC connection to other processes. Instead, the native C++ codes of the rendering engine establish the IPC connection to the *browser process*. When the JavaScript codes call web APIs to exploit various functionalities of the underlying operating systems, the rendering engine forwards the API call to the browser process through the IPC. The browser process handles I/O functionality and returns I/O data to the rendering engine. JavaScript codes usually have callback functions for the I/O requests. Upon receiving the I/O data, a JavaScript runtime inserts the callback functions into the runtime's queue. The event loop of the runtime periodically checks whether the call stack is empty, and if it is, the loop takes a runnable callback function from the queue and then pushes the function into the call stack.

Table I lists the standard web APIs related to I/Os. Each web API has different usage semantics. For instance, the Geolocation API, the MediaStream API, and UIEvent interfaces are handled by global objects. Sensor APIs are accessed through non-global objects; therefore, web applications should create relevant objects by calling constructors. As summarized in Table I, I/O data fall into three types: general, periodic, and media stream. General data indicate a single I/O data response for an I/O request, while periodic data refer to a continuous response to an I/O request. Media stream–type data are video or audio, which is handled as encoded data.

TABLE I. Web APIs for I/O devices.

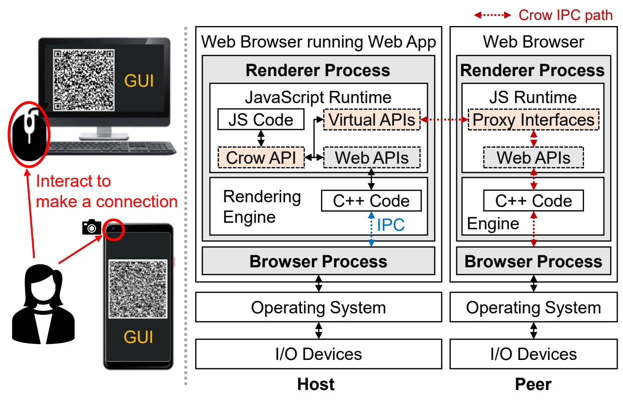
Web API	I/O device	Global	I/O data type
Geolocation API	Global positioning system (GPS)	Yes	General/ Periodic data
MediaStream API	Camera, microphone, speaker	Yes	Media stream
Sensor APIs	Accelerometer, light sensor, etc.	No	Periodic data
UIEvent Interface	Keyboard, mouse, touch, etc.	Yes	General data

C. Challenges

The standard web APIs provide a uniform user experience for various operating systems or browsers. This meta-platform characteristic implies that web applications potentially provide cross-device I/O sharing, in principle, even in heterogeneous environments. Despite this potential, conventional web applications barely support cross-device I/O sharing functionalities because of the non-trivial amount of development efforts. One approach for alleviating developers' efforts is to provide a new API for cross-device I/O sharing. Two key challenges should then be resolved to design such an API.

Challenge 1. The web environment lacks direct support for device discovery. As described in Section II.A, WebRTC does not provide functionality for device discovery. Moreover, web browsers do not support device discovery mechanisms readily used by native applications, such as Bluetooth advertising, Wi-Fi direct, and multicast domain name service (mDNS). Exploiting signaling servers is practically the only way to handle device discovery in web applications. Apart from original web servers, Web developers should implement and maintain signaling servers additionally. Moreover, developers should write codes for their web applications to communicate with the signaling servers. This results in significant development efforts for device discovery. Therefore, a new API is needed to mitigate the development efforts, but its provisioning is challenging due to many practical constraints in device usage.

Challenge 2. JavaScript's inherent characteristics make it difficult for web applications to use other devices' I/O. JavaScript runtimes lack support for general-purpose IPC functionality. As described in Section II.B, a web application needs IPC connections between the renderer process and the browser process to use I/O. Because JavaScript runtimes are unable to establish direct IPC connections with the browser process, the runtimes require support from the C++ codes of the rendering engine to communicate with the browser process. Unfortunately, the rendering engine was developed for single-device environments. The rendering engine of current browsers makes IPC connections only to the browser process in the same browser, and cannot establish direct IPC connections to other device's browser processes. In addition, the JavaScript language does not support direct memory management because JavaScript allocates memory automatically. Previous work on cross-device I/O sharing in Android applications [1]–[3]



(a) Connection Phase (b) I/O Sharing Phase

Fig. 3. The workflow and architecture of the Crow API.

introduced various memory-sharing techniques for synchronizing data. However, due to the inherent limitation of JavaScript in accessing memory areas, ensuring functional consistency is challenging when synchronizing data for cross-device I/O.

III. Crow API Overview

We propose the Crow API to mitigate programming efforts for supporting device discovery and I/O sharing functionality without modifying OSs or browsers. The overall workflow of cross-device I/O-enabled web applications is discussed below, followed by a detailed description of the API functions.

A. *Workflow*

The workflow and overall architecture of the Crow API are illustrated in Fig. 3. The primary device running a web application is defined as the host, and the secondary device delivering I/O functionality to the host is the peer. For cross-device I/O-enabled web applications, developers should implement host and peer applications using the Crow API. The architecture supports multiple peers for a single host; however, to keep the descriptions brief, throughout the paper, we explain the architecture based on a one-to-one host–peer configuration.

The Crow API includes three operational phases: (1) the signaling phase, (2) the I/O sharing phase, and (3) the disconnection phase. First, in the signaling phase, the API displays specific graphical user interfaces (GUIs) on the host and the peer. Users handle the GUIs for the host to discover the peer and establish WebRTC connections to the peer. This GUI-based signaling is used only for the first connection between the host and the peer to minimize the users' engagement. Then, during the I/O sharing phase, users exploit the peer's I/O while the main parts of the web applications run on the host. Finally, in the disconnection phase, the WebRTC connection stops, and web applications perform specific operations defined by the developers. Note that current browsers require web applications to acquire proper permissions for each I/O before using the I/O [9]. Crow API follows the browser's permission policy related to I/O usage.

The Crow API provides solutions to enable the workflow. For the connection phase, the Crow API handles the device discovery issue related to signaling servers. We propose a serverless WebRTC signaling scheme for Crow connectivity.

TABLE II. The Crow API.

Function	Parameters	Return
switchToVirtualAPI()[a]	I/O name	{Host, Peer}Descriptor
switchToOriginalAPI()[a]	I/O name	Void
respondToVirtualAPIRequest()[a]	Void	{Host, Peer}Descriptor
addDisconnectionHandler()[b]	Callback function	Handler ID
removeDisconnectionHandler()[b]	Handler ID	Void

[a] Static Crow API method. [b] {Peer, Host}Descriptor member

```
let peer = await Crow.switchToVirtualAPI("GPS"); // (1) Signaling
navigator.geolocation.getCurrentPosition(…); // (2) I/O Sharing
peer.addDisconnectionHandler((e) => {…}); // (3) Disconnection
Crow.switchToOriginalAPI("GPS");
```

(a) Host

```
let host = await Crow.respondToVirtualAPIRequest();
host.addDisconnectionHandler((e) => {…});
```

(b) Peer

Fig. 4. Sample code with the Crow API.

The GUIs displayed by the Crow API are designed under the connectivity mechanism. Section IV explains the scheme in detail. For the I/O sharing phase, the issues regarding JavaScript's lack of general-purpose IPC and direct memory management are handled adequately. We propose the Crow IPC mechanism through which the Crow API communicates transparently with other devices' browser processes. The Crow IPC is detailed in Section V.

B. *Developer API*

The Crow API functions are summarized in Table II. Fig. 4 shows a sample code of a web application that uses the Crow API to retrieve GPS data from a peer. In this example, the host and the peer are a desktop and a smartphone, respectively. A key goal of the Crow API design is to provide simplicity of usage. By simply adding a few lines of the Crow API codes, web developers can provide device discovery and I/O sharing functionalities for their web applications.

Signaling Phase. The Crow API provides two functions for the signaling phase: switchToVirtualAPI() and respondTo VirtualAPIRequest(). On the host, web applications call switchToVirtualAPI() and pass an argument (i.e., the I/O name to use from the peer) to the function. On the peer, respondToVirtualAPIRequest() is used to respond to the host's request. The signaling process caused by these functions depends on whether it is the first signaling between the host and the peer. If it is, the functions display the GUI for making a WebRTC channel on the host and the peer. Otherwise, the host and the peer make WebRTC channels without the GUI, which is possible with the channel embedding described in Section IV.B.

I/O Sharing Phase. Cross-device I/O-enabled web applications use the same standard web API functions to get the I/O data from the peer, even after the Crow API is applied to the applications. For example, web applications originally use the getCurrentPosition() function to obtain the host's GPS data. Web applications use the same function to obtain the peer's GPS data. The difference is whether the Crow API functions are called during the connection phase. To transparently access the peer's GPS data, the Crow API introduces *virtual APIs*. The API calls for the peer's I/O are forwarded to the virtual API objects instead of the original web APIs. The virtual APIs communicate with the peer's browser processes through the Crow IPC.

TABLE III. VARIOUS OPTIONS FOR SERVERLESS SIGNALING.

	Sender				Receiver			
	BT	***NFC***	***QR***	***MousePath***	***BT***	***NFC***	***QR***	***MousePath***
Desktop			✓	✓				✓
Laptop	✓[a]		✓	✓	✓[a]		✓	
Tablet	✓[a]	✓[a]	✓	✓	✓[a]	✓[a]	✓	
Smartphone	✓[a]	✓[a]	✓	✓	✓[a]	✓[a]	✓	

[a] Device discovery is possible on native apps; while not on web apps.

Disconnection Phase. The connection between the host and the peer terminates upon the user's intention or network intermittency. To handle cases where web applications wish to stop the connections, the Crow API provides the switchTo OriginalAPI() function. For dealing with unintended network intermittency, the Crow API provides two functions: add DisconnectionHandler() and removeDisconnectionHandler().

IV. CROW CONNECTIVITY MECHANISM

The Crow connectivity mechanism enables web applications to discover other devices without a signaling server, and establish WebRTC connections directly to the devices. For the serverless WebRTC signaling, we propose a signaling scheme through which web applications exchange SDPs. Moreover, we provide techniques for optimizing the connectivity mechanism.

A. Serverless Signaling Scheme

Establishing WebRTC connections requires web applications to exchange SDPs. The Crow API's serverless signaling scheme is specifically designed to allow web applications to discover other devices and exchange SDPs without signaling servers. The signaling scheme should employ device discovery and data transmission functionalities for web applications. The scheme also aims to accommodate a wide range of computing devices, such as desktops, laptops, tablets, and smartphones.

To develop the serverless signaling scheme, we evaluated various options listed in Table III. The methods broadly fall into two categories: radio frequency (RF)-based schemes and display-based schemes. The RF-based approach includes Bluetooth (BT) and near-field communication (NFC), which are commonly used in native applications. Unfortunately, the RF-based methods are not suitable for the Crow API's signaling scheme because the relevant APIs do not support low-level operations such as device discovery—an essential feature for signaling. Furthermore, desktop computers generally do not support RF-based schemes. The display-based schemes send data by encoding the data and exposing the encoded data on display screens. QR codes and MousePath [8] belong to this category. QR codes are widely used for display-to-camera communication. MousePath was proposed for display-to-mouse communication. The scheme sends data by displaying the movement of textures on the display screen and receives the data by sensing movement with the optical mouse. The display-based schemes have advantages compared to the RF-based schemes. First, the display-based approach is fully compatible with web applications. Web applications can use and control screens, cameras, and mouses with virtually no constraints. Second, the display-based approach does not necessitate a complicated method for device discovery, such as Bluetooth advertising. Device discovery of the display-based approach just relies on users—the devices exchange data under physical proximity.

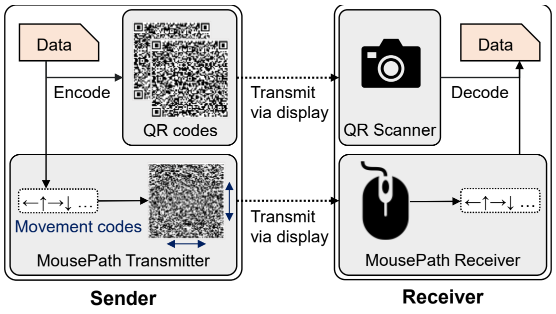


Fig. 5. The data transmission of the Crow API's signaling scheme.

Third, the display-based approach has broad coverage of devices because computing devices have display screens as basic output devices.

The Crow API's signaling scheme utilizes the display-based approach. Fig. 5 illustrates the data transmission of the Crow API's signaling scheme. The data transmission which is necessary for SDP exchange consists of a sender and a receiver. As shown in Table III, each method does not provide full coverage, especially in terms of receivers. The signaling scheme, therefore, does not rely solely on a single communication method; instead, the proposed scheme combines the two methods, QR codes and MousePath, to compensate for the coverage constraints of each method. When users wish to send data, the sender encodes the data into the QR codes or the MousePath codes and displays the codes on the screen. The receiver acquires the codes with the camera or the optical mouse, and decodes the codes into the original data.

This combination makes it possible for the serverless signaling scheme to support various peer-to-peer configurations. Note that WebRTC connection needs bidirectional SDP exchanges, as described in Section II.A. The computing devices listed in Table III fall into two groups: (1) mouse-equipped devices, such as desktops, and (2) camera-equipped devices such as laptops, tablets, and smartphones. For connections between two mouse-equipped devices, both devices use MousePath to transmit data. For connections between two camera-equipped devices, the devices exchange data through QR codes. For connections between mouse- and camera-equipped devices, camera-equipped devices send data through MousePath, while mouse-equipped devices transmit data via QR codes.

B. Optimization

The display-based approach requires end-users to cope with cameras or optical mouses by themselves. Considering the amount of SDP data and data throughput of the display-based schemes, users may have to spend a substantial amount of time making a WebRTC connection, leading to a poor user experience. We propose three optimization techniques for minimizing user engagement: channel embedding, protocol data reduction, and prioritization. Fig. 6 shows the overall flow of the connectivity mechanism with the proposed techniques.

Channel Embedding. The idea of the channel embedding is to embed WebRTC media channels or other data channels into WebRTC data channels, as described in Fig. 6. There are two types of WebRTC channels: data channels and media channels. Data channels are used for exchanging text messages between

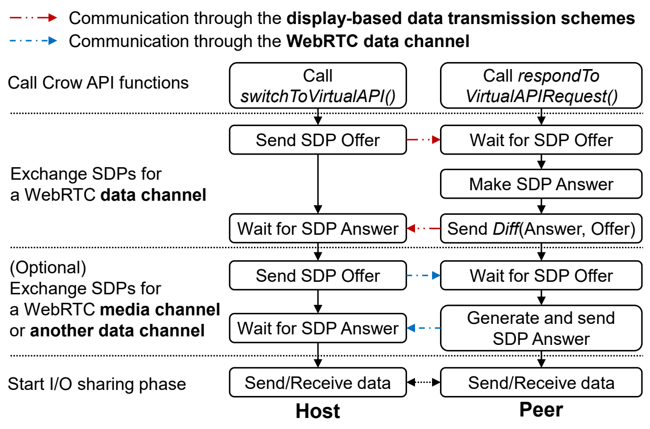

Communication through the display-based data transmission schemes
Communication through the WebRTC data channel
Call Crow API functions
Call switchToVirtualAPI()
Call respondTo VirtualAPIRequest()
Exchange SDPs for a WebRTC data channel
Send SDP Offer
Wait for SDP Offer
Make SDP Answer
Wait for SDP Answer
Send Diff(Answer, Offer)
(Optional) Exchange SDPs for a WebRTC media channel or another data channel
Send SDP Offer
Wait for SDP Offer
Wait for SDP Answer
Generate and send SDP Answer
Start I/O sharing phase
Send/Receive data
Send/Receive data
Host
Peer

Fig. 6. The flow of the Crow connectivity mechanism.

devices, while media channels are used for sharing videos or audio. The SDPs for media channels include various types of information, such as codecs, bitrates, etc.; thus, the length of the SDPs for the media channels is significantly longer than that of data channels. When web applications want to make media channels, the Crow connectivity mechanism first establishes a WebRTC data channel through the display-based data transmission schemes, and then the SDPs for the media channel are transmitted through the data channel. With this scheme, the amount of data sent is significantly reduced when making media channels, and so is user involvement. Channel embedding is also used when the host and the peer make additional channels for I/O sharing. If the host and the peer already have a data channel between them, the extra call to switchToVirtualAPI() does not require user involvement for the display-based signaling. The host and the peer make WebRTC channels via the data channel established by the first call to switchToVirtualAPI().

Protocol Data Reduction. Protocol data reduction minimizes the time taken for the Crow API's signaling scheme to send data by removing the redundant parts between an SDP offer and an SDP answer. The offer and the answer contain similar data because they are based on the same structure [10]. After receiving the SDP offer from the host, the peer sends the data to the host with the redundant parts removed in the protocol.

Prioritization between Data Transmission Schemes. For connections between mouse- and camera-equipped devices, where both QR codes and MousePath are used, the Crow connectivity mechanism sets priorities between the QR codes and MousePath. The QR code method is simpler and easier for end-users to handle than MousePath. In addition, the length of the SDP answers becomes shorter than that of the offers with the protocol data reduction scheme. Therefore, to make the most of the protocol data reduction, mouse-equipped devices should send the SDP offers through QR codes, and camera-equipped devices should send the SDP answers via MousePath. However, there are cases where the host and the peer are camera- and mouse-equipped devices, respectively, as shown in Fig. 1(b). Thus, the Crow API allows both hosts and peers to start the device discovery process.

V. Crow IPC

In this section, we describe the Crow IPC mechanism, which addresses JavaScript's inability to establish direct IPC connections to other devices' browser processes. We also describe how Crow IPC ensures functional consistency of cross-device I/O-enabled web applications. Fig. 7 illustrates the overall view of Crow IPC.

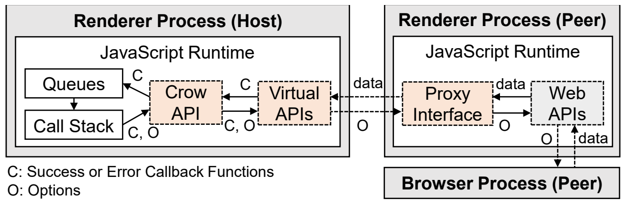


Fig. 7. Crow IPC workflow.

A. Proxy Interface

A proxy interface is a JavaScript object that relays an IPC connection to the peer's browser process. For Crow IPC, the Crow API exploits the peer's runtime through a proxy interface. When a web application tries to use the peer's I/O, the virtual API forwards the I/O-related web API function to the peer (the *function-forwarding phase*). Upon receiving the forwarded API function, the proxy interface calls the web API functions on behalf of the host. After the peer's browser process returns I/O data to the proxy interface through the web APIs, the proxy interface forwards the I/O data to the host's virtual API (the *data-receiving phase*). With the proxy interface, the host's JavaScript runtime can communicate with the peer's browser process.

B. Ensuring Functional Consistency

Crow IPC provides two mechanisms to ensure functional consistency: reference-free function forwarding and cross-device data synchronization.

Reference-free Function Forwarding. Reference-free function forwarding is used when a virtual API forwards the web API functions from the host to the peer. The key concept is to exchange data free of side effects between the host and the peer, while the functions that are likely to have side effects remain on each device. The web API functions typically have three kinds of parameters: callback functions for success, callback functions for error, and non-function values such as options and constraints. Except for the non-function values, the parameters are likely to cause side effects because JavaScript functions are not purely functional; they may alter variables or states outside the functions. Therefore, a virtual API should not forward callback functions to the peer. When a virtual API delegates web API functions to the peer, the virtual API sends only the name of the web API function and the option values.

Cross-device Data Synchronization. After functions are forwarded, the host's virtual APIs receive I/O data from the peer's proxy interface through cross-device data synchronization. The overall concept is illustrated in Fig. 8. Crow IPC provides three kinds of data synchronization methods—one for each I/O data type discussed in Table I. For general data, the proxy interface returns I/O data to the virtual APIs through messages. Note that callback functions remain in the host during the function-forwarding phase. The virtual APIs maintain a map data structure to record which callback function is required for I/O data. Upon getting I/O data via messages, the virtual APIs look up the relevant callback function in the map

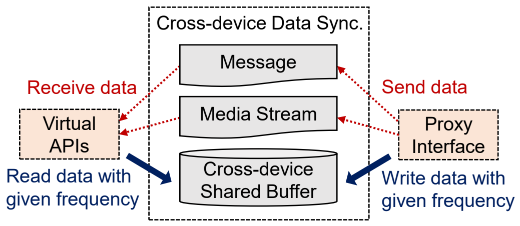


Fig. 8. Cross-device data synchronization.

and insert the callback function into the queues in a JavaScript runtime. The runtime then executes the callback function when the call stack of the runtime is empty.

For periodic data, Crow IPC uses a cross-device shared buffer to share I/O data between the virtual APIs and the proxy interface. The synchronization process follows a producer–consumer pattern. After calling the web APIs, the virtual APIs try to read I/O data from the shared buffer with the given frequency. If the buffer is empty, the virtual APIs' reading tasks wait until the peer writes the data to the buffer; if not, the virtual APIs read the data and execute the callback function for the I/O. As the JavaScript is single-threaded, the reading tasks should not use busy waiting. Instead, the virtual APIs have queues in which the reading tasks are inserted and wait asynchronously. Meanwhile, the peer's proxy interface tries to write I/O data to the buffer and has a queue for the writing tasks' asynchronous waiting.

For media streams such as video and audio, it is well-known that sharing encoded data is much more efficient than sharing raw data. Thus, we utilize standard WebRTC streaming to share the media streams. When a web application requires streams from the peer's I/O, the Crow API establishes the WebRTC stream connection between the host and the peer. The callback function for the I/O waits in the host's JavaScript runtime until the WebRTC stream comes from the peer, similar to the message method for general data. WebRTC streams video and audio at the best quality based on the Google congestion control algorithm [11]; thus, the Crow API efficiently provides streaming-based I/O sharing between multiple devices.

VI. Implementation

We implemented the Crow API as a JavaScript library for application developers to readily import into their applications. Additionally, we developed the Crow extension program, which automatically applies the Crow API to existing web applications.

A. *The Crow API Library*

Crow Connectivity Mechanism. We implemented the Crow connectivity mechanism as part of the Crow API. When web applications call switchToVirtualAPI() or respondTo VirtualAPIRequest(), the Crow API displays GUIs to send and receive QR codes or MousePath. Open-source JavaScript codes were used to implement QR-based data transmission. For MousePath, we implemented the core as described in the paper [8] and employed open-source JavaScript codes for the required Reed-Solomon error correction. When web applications send data through MousePath, error correction codes are attached to the original data. The length of the error correction is set at 20% of the original data. As described in Section IV.B, the Crow API also allows the peers to start device discovery by calling the switchToVirtualAPI() function, and the hosts should respond to the requests with the respondToVirtualAPIRequest() function.

Crow IPC. The Crow API library is compatible with web APIs for I/Os listed in Table I. We implemented the Crow API's VirtualAPI objects according to the way each API works. As stated in Section II.B, some web APIs are invoked via global objects, while others are not. For global object–based web APIs, such as the MediaStream API and the Geolocation API, the VirtualAPI object has the same structure as the original global interfaces. For Sensor APIs, which do not have global objects but global constructors, VirtualAPI objects are generated by constructor calls. For keyboard events, the DOM waits for them without explicit function calls to turn the keyboard on or off. When users click or touch DOM nodes related to keyboard events, the web application obtains keyboard events from the peer if the virtual API for the keyboard is set to the current interface. For pointing devices, including the mouse, touch, stylus pen, etc., the library forwards pointing events, such as click/touch and drag/swipe, from the peer to the host with the relative position where the event occurs on the screen. Finally, the Crow API supports display by exploiting WebRTC's screen sharing. There are no web API functions to turn the display on or off; thus, display sharing starts as soon as a web application calls the switchToVirtualAPI() function to set a virtual API object as the current interface.

B. *The Crow Extension Program*

Although the Crow API library helps application developers easily exploit cross-device I/O sharing functionality, existing web applications do not provide I/O sharing unless they are modified with the library. We implemented the Crow extension program, which provides cross-device I/O sharing functionality to existing web applications by automatically injecting the Crow API. The host runs the extension program in the browser. The peer runs a proxy web application that delivers the I/O data to the host. The Crow extension program provides two key functionalities: I/O selection and automatic code injection. The extension program presents a popup GUI that asks users which device to use for each I/O before running the web application. Then, the Crow extension program injects custom code into the web application during the loading process. We developed the Crow extension program as a Chrome extension; thus, the extension program works on Chromium-based browsers. The Chrome browser does not support the extension on Android, but the Kiwi browser does. As the WebExtension API is being standardized, this implementation is expected to work on all browsers soon.

VII. Evaluation

We evaluated the Crow connectivity mechanism in terms of the amount of data required for the WebRTC connection. We also compared the I/O sharing functionality of the Crow API with other solutions in real environments and measured the performance with diverse I/O devices.

A. *Crow Connectivity Mechanism*

To evaluate the effectiveness of two optimization techniques (channel embedding and protocol data reduction), we measured the length of the data sent via the Crow API's data transmission scheme while establishing a WebRTC video channel. For the

TABLE IV. CROSS-DEVICE I/O SHARING OF THE CROW API AND PREVIOUS SOLUTIONS.

I/O	Web app	Crow	Air Play	Chrome cast	Droid Cam	WO Mic	Rio	PMC	M+	M2	Tap
Camera	Zoom, WebEx, Google Duo, Google Meet, Talky, Whereby	✓			✓		✓	✓	✓	✓	✓
Microphone	Zoom, WebEx, Google Duo, Google Meet, Talky, Whereby	✓				✓	✓			✓	✓
GPS	Google Maps, OpenStreetMap, Bing Maps, Waze, Airbnb	✓						✓		✓	✓
Motion and light sensors	(Demo web applications)	✓					✓[a]	✓[a]	✓[a]	✓	✓
Keyboard	Google Docs, Wikipedia	✓								✓	✓
Pointer (Mouse, touch. stylus)	Google Keep, Chrome Canvas, Sketch.io, Slither.io	✓								✓[b]	✓[b]
Speaker	YouTube Music, Soundcloud	✓	✓	✓			✓	✓		✓	✓
Display	YouTube, Netflix, Twitch, TED	✓	✓	✓						✓	✓

[a] Supports only accelerometers among various sensors. [b] Supports drag or swipe events, but cannot share the movement of mouse cursors.

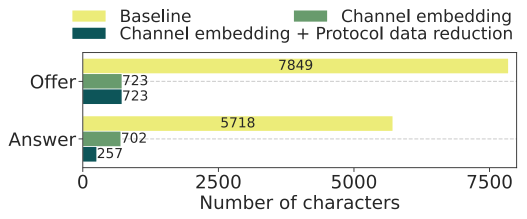


Fig. 9. Message length for establishing a WebRTC video channel between two devices.

measurement, we used the desktop-to-tablet configuration, and the devices were connected to the same router. We set the desktop and the tablet as the host and the peer, respectively.

Fig. 9 shows the measurement result. The baseline in the figure refers to the case where no optimization techniques are applied. For the baseline case, the length of the SDP offer and the answer are 7,849 and 5,718 characters, respectively. QR codes containing a large amount of data are difficult to recognize with cameras on general mobile devices due to the codes' complexity. Thus, previous QR-based solutions for the serverless WebRTC connection [12], [13] use multiple QR codes to relay the SDPs. In the case of MousePath, it takes more than 450 s to send the 5,718-character SDP answer, which is too long for users to wait for. We calculated the time under the assumption that MousePath transmits 15 characters a second on 60-Hz screens and requires error correction codes, 20% of the original data. As shown in the figure, the proposed optimization techniques efficiently reduce the length of SDPs. For the channel embedding, the length of the SDP offer and answer decrease to 723 and 702 characters, respectively. If the connectivity mechanism employs both techniques, the length of the SDP answer is reduced to 257 characters. In this case, the connectivity mechanism needs only one or two QR codes to transfer an SDP offer. It takes only 20 s to forward an SDP answer via MousePath, even taking into account the error correction codes. This results in a 95.5% reduction compared to the baseline. The Crow connectivity mechanism indeed provides a practical solution.

B. *I/O Sharing Functionality*

Cross-I/O Coverage. To confirm that the Crow API can provide cross-device I/O sharing, we evaluated whether it worked well on popular web applications and compared the Crow API's I/O coverage with that of other solutions. For the comparison, we selected various cross-device I/O sharing solutions, including Android-based native solutions for general I/O sharing, such as Rio [1], Personal Mobile Cloud (PMC) [2], Mobile Plus (M+) [3], M2 [4], and Tap [14].

The results of the cross-I/O coverage are summarized in Table IV. The evaluation was conducted on a host-peer configuration, where each solution best supported cross-device I/O sharing. For the Crow API, we used a Windows 10-based laptop for the host and the Android 11-based Pixel 2 XL for the peer. Chrome version 92 was used for the evaluation. The "I/O" column in the table lists the types of I/Os tested in this experiment. We evaluated the functionality of the Crow API with the sample web applications listed in the "Web app" column. For motion and light sensors, we used demo web applications we wrote for the test because we were unable to find popular web applications that use Sensor APIs.

As shown in the table, the Crow API supported the most diverse types of I/Os. AirPlay [15] and Chromecast [16] were developed for sharing audio and video, DroidCam [17] for sharing cameras, and WO Mic [18] for sharing microphones. These solutions support limited I/O as they were developed for sharing specific I/Os. However, the Android-based I/O sharing solutions show wider I/O coverage than the I/O-specific solutions. Android-based solutions aim to provide general I/O sharing by modifying operating systems or providing an SDK for developing cross-device applications. Only M2 and Tap native applications support as many I/Os as the Crow API. This result supports the generality of the Crow API in providing cross-device I/O sharing even as a web application.

Heterogeneity. We evaluated the functionality of the Crow API on diverse platforms and compared the results with those for other solutions. For the evaluation, we selected five platforms (Android, iOS, Windows, macOS, and Linux) and used the platforms as hosts and peers. For the devices, we used a Pixel 2 XL smartphone for Android, an iPad Pro tablet for iOS, an i5 laptop computer for Windows, an iMac desktop for macOS, and an i7 desktop computer for Linux. For all cases except the Android host and the iOS host, Chrome version 92 with the Crow extension program was used for the evaluation. For the Android host, we used the Kiwi browser, a modified version of Chrome that supports extensions on Android. For the iOS host, we modified web applications with the Crow API because no browsers support the extension on iOS. For the iOS host and iOS peer, the evaluation was performed on the Safari browser, which is the main browser for iOS. Note that the Safari browser is planned to support extension programs built with standard WebExtension API [19]; therefore, we expect the proposed Crow extension program will work on iOS in the near future.

Table V summarizes the results, showing that the Crow API outperformed other solutions in supporting cross-device I/O

TABLE V. Cross-device I/O sharing on diverse platforms.

Platform	Crow	Air Play	Chrome cast	Droid Cam	WO Mic	Rio	PMC	M+	M2	Tap
Android host	✓		✓			✓	✓	✓	✓	✓
iOS host	✓[a]	✓								✓
Windows host	✓		✓	✓	✓					
macOS host	✓									
Linux host	✓			✓		✓				
Android peer	✓		✓	✓	✓	✓	✓	✓	✓	✓
iOS peer	✓[b]	✓		✓					✓	✓
Windows peer	✓									
macOS host	✓	✓								
Linux host	✓					✓				

[a] The Crow extension does not work, but the Crow API works.
[b] The Crow API cannot use Sensor APIs due to the lack of browser support.

sharing on diverse platforms. Previous solutions are inherently limited to specific platforms because the solutions are for native applications. AirPlay and Chromecast work only for the Apple (iOS/macOS) and Google (Android) platforms, respectively, for sharing display and audio. DroidCam and WO Mic were developed for a specific host-peer configuration: desktop host and mobile peer. Rio, PMC, M+, M2, and Tap are Android-based solutions. M2 and Tap are designed to work on iOS but do not work as a general solution on other platforms. Compared to the previous solutions, the Crow API provides fully heterogeneous cross-device I/O sharing due to its meta-platform characteristics.

There were a few cases in which the Crow API could not provide cross-device I/O sharing for specific I/Os, primarily because of the browsers' limitations. For the iOS peer, the Safari browser and other browsers, including Chrome for iOS, do not support Sensor APIs; thus, cross-device I/O sharing cannot be utilized for sensors. Meanwhile, the iOS host can access the other device's sensors through the Crow API, although iOS does not support Sensor APIs. This is because web API functions are executed not on the host but on the peer where browsers support Sensor APIs unless the peer is not iOS. As the Sensor APIs are being standardized, we hope that the Safari browser will support Sensor APIs so that web applications will be able to provide cross-device I/O sharing functionality even on an iOS peer.

C. Performance

I/O Sharing Overhead. We evaluated the performance of the Crow API in providing cross-device I/O sharing. We first assessed the overhead of the cross-device function call by measuring the time from when the web API functions were called until the first I/O data arrived. In this evaluation, the host was the Windows 10-based laptop with an Intel i5-8265U CPU, and the peer was the Android 10-based Pixel 2 XL smartphone. The time measurement was performed in the 50-Mbps Wi-Fi environment, which is a typical environment for users to experience cross-device I/O sharing with their devices.

Fig. 10 shows the measurement results of the Crow API overhead in providing cross-device I/O sharing functionality for each I/O. Each bar in the figure represents the average time of 100 measurements. The overhead was almost constant, with an average of 86.3 ms, except for the camera. This overhead is small enough to barely affect the web application's functionalities and the overall user experience. We further analyzed the cause of the overhead and found that it was mostly

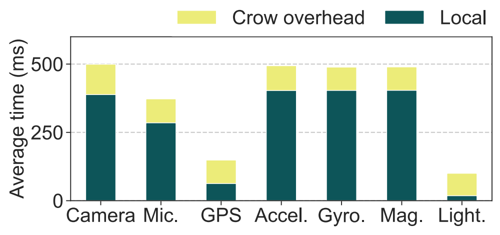


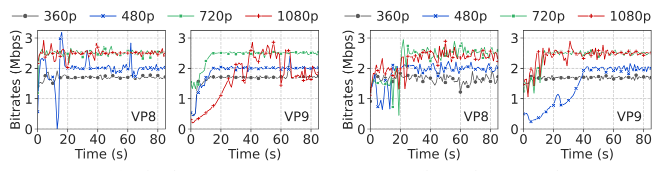
Fig. 10. Overhead for calling the peer's web API functions with the Crow API.

caused by the round-trip time (RTT) of the WebRTC data channel. The measured RTT was about 81 ms, similar to results found in previous research [20]. The actual processing time for the Crow API code was at most 5 ms, which has a trivial effect on the performance. For the camera, the overhead was about 111 ms which is relatively larger than that of the other I/O. This additional overhead comes from establishing the WebRTC stream connection, which is necessary for the Crow API to share stream-based I/O between devices. Meanwhile, the overhead for the microphone, another stream-based I/O, is similar to that of the other I/O because the Crow API establishes the WebRTC stream connection while waiting for the peer's data which come through the WebRTC data channel. Establishing the WebRTC stream connection for audio took less time than the WebRTC data channel's RTT between the host and the peer.

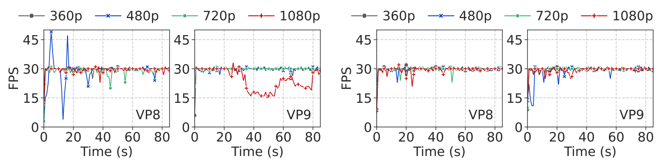
Other I/Os (such as the keyboard, speaker, and display) are not included in the figure because there are no comparison baselines: there are no web API functions to turn these I/Os on. For these I/Os, the performance of the Crow API depends heavily on the WebRTC's performance. Because the WebRTC data channel is currently the only option for exchanging data between web applications without a third-party server, this overhead is unavoidable for the Crow API. Therefore, the overall performance of the proposed API depends on the implementation quality of WebRTC's data channel.

Camera Sharing. We specifically conducted a performance evaluation of camera sharing because video-based applications are commonplace. Normally, cross-device camera sharing uses a non-trivial amount of resources; thus, previous solutions, especially Android-based solutions [1]–[3], showed significant performance degradation for sharing cameras. The existing solutions share raw camera data with the memory-sharing approach; in contrast, the Crow API uses compressed data with WebRTC. In this evaluation, we used the same host–peer configuration as previously. In addition, we used another peer, the Galaxy S9 Plus, which provides a better computation capability than the Pixel 2 XL. We set the peer's camera resolution to 360p, 480p, 720p, and 1080p, and the camera's frames per second (FPS) to 30 FPS. We used the VP8 and VP9 codecs, which are default codecs used by WebRTC. The VP9 codec is known to compress videos more efficiently than the VP8 codec but demands more CPU.

Fig. 11 shows the bitrate traces of the shared videos measured up to 80 s after cross-device camera sharing starts. As shown in the figure, the bitrates stabilize after 40 s and converge to specific values. A previous study on WebRTC video streaming [11] reported fixed 2.5-Mbps bitrates at 720p or higher resolutions on desktop computers. The study also showed that mobile devices were not able to stream 2.5-Mbps videos due to the devices' low performance. However, we experimentally



(a) Pixel 2 XL (b) Galaxy S9 Plus
Fig. 11. Bitrate traces.



(a) Pixel 2 XL (b) Galaxy S9 Plus
Fig. 12. FPS traces.

found that current mobile devices transmit 2.5-Mbps videos without any problems because of the performance improvement in newer mobile devices. We confirmed that the output resolutions are not degraded for almost all cases. Only in the case of the Pixel 2 XL with the VP9 codec, the 1080p resolution decreased to 540p because the Pixel 2 XL does not have enough performance to support the VP9 codec at a resolution of 1080p. We further evaluated FPS for each case. Fig. 12 shows the results. We found that the FPS traces converge to 30 for almost all cases, aside from the case of the Pixel 2 XL with the 1080 resolution and the VP9 codec.

VIII. Related Work

Previous researchers have tried to exploit the potential of using multiple devices for improved user experiences. In particular, various techniques have been proposed for distributing GUIs or sharing I/Os on multiple devices. Many practical cross-device solutions have been developed especially for remote desktop environments, such as Virtual Network Computing (VNC) [21], TeamViewer [22], AnyDesk [23], and Chrome Remote Desktop [24]. The tools exploit a specific I/O set—display, mouse, keyboard, and speaker—that is necessary for remote desktop processing. The tools mandate the use of display sharing, even if a user wishes to utilize only a remote keyboard or other simple I/Os. Clearly, these tools are limited in providing general-purpose cross-device I/O sharing.

In addition, various solutions for cross-device GUI distribution to multiple devices have been proposed. Chromecast [16] allows users to cast video- and audio-related GUIs of applications to other devices. CollaDroid [25] and UIWear [26] help developers modify their Android applications, whose GUIs are originally designed for single-device environments, to cross-device GUI versions. Panelrama [27], Liquid.js [28], and AdaM [29] help developers implement web applications that provide cross-device GUI functionality. Recently, work has been conducted to provide cross-device GUI functions without reauthoring by developers. FLUID, FLUID-XP, and PRUID [30]–[32] enable users to distribute Android applications' GUIs to multiple devices. XDBrowser [33], [34] provides a cross-device GUI solution for web applications without reauthoring. Cross-device GUI distribution provides interesting use cases, but the technique focuses on fine-grained display sharing, which is different from the goal of the Crow API, which is general I/O sharing.

To provide cross-device I/O sharing functionality, many efforts have been made especially for native applications. The solutions fall into three categories. First, applications have been developed for sharing specific I/Os. IP Webcam [35] and DroidCam [17] allow users to employ a smartphone's camera on Windows desktop computers, and WO Mic [18] allows users to use smartphones' microphones. These applications lack generality in providing cross-device I/O sharing. Second, the OS-level approach has been proposed for cross-device I/O sharing. Rio [1] implements cross-device I/O sharing through a split-stack kernel architecture that intercepts Linux kernel operations. Personal Mobile Cloud [2], Mobile Plus [3], and Jong et al. [36] introduced solutions that work at the Android framework layer. M2 [4] achieved cross-device I/O sharing between iOS and Android applications by leveraging I/O abstractions. These OS-specific system-level solutions are unable to run on diverse platforms, leading to the platform-dependency problem. Finally, researchers have proposed APIs or SDKs that help developers adopt cross-device I/O sharing functionality. Tap [14] employs a cross-device I/O API for Android and iOS applications. However, native applications developed with the API should run on specific platforms such as Android or iOS; thus, this API approach still cannot solve the platform-dependency issue.

Overall, previous solutions for cross-device I/O sharing have limited I/O support or platform-dependency problems. In contrast, the Crow API provides cross-device I/O sharing running on diverse platforms by exploiting web applications' meta-platform characteristics. In fact, there was a solution for cross-device I/O sharing on web applications, called Gibraltar [37]. Gibraltar exposed I/O devices to web pages via an HTTP-enabled device server that interacts with I/O devices on behalf of web pages. Gibraltar was a pre-WebRTC system; thus, it required the host and peers to run device servers and know each server's IP address. On the other hand, Crow API is compatible with modern web standards, such as WebRTC and web APIs, and provides I/O sharing functionalities without any additional server programs.

IX. Conclusion

To the best of our knowledge, the Crow API is the first attempt to provide general-purpose cross-device I/O sharing in web applications. Based on the meta-platform characteristics of web applications, the Crow API addresses the platform-dependency issue of native application-based solutions for cross-device I/O sharing. Recently, new web APIs have been proposed and standardized for various hardware, such as the Bluetooth API, NFC API, and Human Interface Device (HID) API. The Crow API is designed to be extensible to these new APIs. We hope that it becomes a standard for web APIs and opens up a new direction for general cross-device I/O sharing for web applications.

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] Ardalan Amiri Sani, Kevin Boos, Min Hong Yun, and Lin Zhong, "Rio: A System Solution for Sharing i/o between Mobile Systems," In Proc. 12th Annu. Int. Conf. Mob. Syst. Appl. Serv. (MobiSys '14), 2014, pp. 259–272.

[2] Yong Li and Wei Gao, "Interconnecting heterogeneous devices in the personal mobile cloud," In IEEE INFOCOM 2017 - IEEE Conf. Comput. Commun., 2017, pp. 1–9.

[3] Sangeun Oh, Hyuck Yoo, Dae R Jeong, Duc Hoang Bui, and Insik Shin, "Mobile Plus: Multi-device Mobile Platform for Cross-Device Functionality Sharing," In Proc. 15th Annu. Int. Conf. Mob. Syst. Appl. Serv. (MobiSys '17), 2017, pp. 332–344.

[4] Naser AlDuaij, Alexander Van't Hof, and Jason Nieh, "Heterogeneous Multi-Mobile Computing," In Proc. 17th Annu. Int. Conf. Mob. Syst. Appl. Serv. (MobiSys '19), 2019, pp. 494–507.

[5] "Accessing hardware devices on the web," https://web.dev/devices-introduction/.

[6] "Progressive web apps," https://web.dev/progressive-web-apps/.

[7] "WebRTC," https://webrtc.org/.

[8] Zhiwei Wang, Qianyi Huang, Yihui Yan, Haitian Ren, Yizhou Zhang, and Zhice Yang, "MousePath: Enhancing PC Web Pages through Smartphone and Optical Mouse," In 2021 IEEE Int. Conf. Pervasive Comput. Commun. (PerCom 2021), 2021, pp. 1–7.

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

[10] "Annotated example SDP for WebRTC," https://tools.ietf.org/id/draft-ietf-rtcweb-sdp-08.html#rfc.section.3.

[11] Bart Jansen, Timothy Goodwin, Varun Gupta, Fernando Kuipers, and Gil Zussman, "Performance Evaluation of WebRTC-based Video Conferencing," ACM SIGMETRICS Perform. Eval. Rev., vol. 45, no. 3, pp. 56–68, 2018.

[12] "GitHub - TomasHubelbauer/qr-channel," https://github.com/TomasHubelbauer/qr-channel.

[13] "Serverless WebRTC using QR codes," https://franklinta.com/2014/10/19/serverless-webrtc-using-qr-codes/.

[14] Naser AlDuaij and Jason Nieh, "Tap: An App Framework for Dynamically Composable Mobile Systems," In Proc. 19th Annu. Int. Conf. Mob. Syst. Appl. Serv. (MobiSys '21), 2021, pp. 336–349.

[15] "AirPlay - Apple," https://www.apple.com/airplay/.

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

[17] "DroidCam," https://www.dev47apps.com/.

[18] "WO Mic - FREE microphone," https://wolicheng.com/womic/.

[19] "Apple brings Safari web browser extensions to iPhone and iPad with iOS 15 - 9to5Mac," https://9to5mac.com/2021/06/07/apple-brings-safari-web-browser-extensions-to-iphone-and-ipad-with-ios-15/.

[20] Kiran Jadhav, D. G. Narayan, and Mohammed Moin Mulla, "Performance Evaluation of WebRTC for Peer-to-Peer Communication," Lect. Notes Electr. Eng., vol. 735, pp. 455–466, 2021.

[21] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper, "Virtual network computing," IEEE Internet Comput., vol. 2, no. 1, pp. 33–38, 1998.

[22] "TeamViewer: The Remote Desktop Software," https://www.teamviewer.com/en-us/.

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

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

[25] Jiahuan Zheng, Xin Peng, Jiacheng Yang, Huaqian Cai, Gang Huang, Ying Zhang, et al., "CollaDroid: Automatic Augmentation of Android Application with Lightweight Interactive Collaboration," In Proc. 2017 ACM Conf. Comput. Support. Coop. Work Soc. Comput. (CSCW '17), 2017, pp. 2462–2474.

[26] Jian Xu, Qingqing Cao, Aditya Prakash, Aruna Balasubramanian, and Donald E Porter, "UIWear: Easily Adapting User Interfaces for Wearable Devices," In Proc. 23rd Annu. Int. Conf. Mob. Comput. Netw. (MobiCom '17), 2017, pp. 369–382.

[27] Jishuo Yang and Daniel Wigdor, "Panelrama: Enabling Easy Specification of Cross-Device Web Applications," In Proc. SIGCHI Conf. Hum. Factors Comput. Syst. (CHI '14), 2014, pp. 2783–2792.

[28] Andrea Gallidabino and Cesare Pautasso, "The Liquid User Experience API," In Companion Proc. Web Conf. 2018 (WWW '18), 2018, pp. 767–774.

[29] Seonwook Park, Christoph Gebhardt, Roman Rädle, Anna Maria Feit, Hana Vrzakova, Niraj Ramesh Dayama, et al., "AdaM: Adapting Multi-User Interfaces for Collaborative Environments in Real-Time," In Proc. 2018 CHI Conf. Hum. Factors Comput. Syst. (CHI '18), 2018, pp. 1–14.

[30] Sangeun Oh, Ahyeon Kim, Sunjae Lee, Kilho Lee, Dae R. Jeong, Steven Y. Ko, et al., "FLUID: Flexible User Interface Distribution for Ubiquitous Multi-Device Interaction," In 25th Annu. Int. Conf. Mob. Comput. Netw. (MobiCom '19), 2019, pp. 1–16.

[31] Sunjae Lee, Hayeon Lee, Hoyoung Kim, Sangmin Lee, Jeong Woon Choi, Yuseung Lee, et al., "FLUID-XP: Flexible User Interface Distribution for Cross-Platform Experience," In Proc. 27th Annu. Int. Conf. Mob. Comput. Netw. (MobiCom '21), 2021, pp. 762–774.

[32] Menglong Cui, Mingsong Lv, Qingqiang He, Caiqi Zhang, Chuancai Gu, Tao Yang, et al., "PRUID: Practical User Interface Distribution for Multi-surface Computing," In Proc. - Des. Autom. Conf. (DAC '21), 2021, pp. 679–684.

[33] Michael Nebeling and Anind K Dey, "XDBrowser: User-Defined Cross-Device Web Page Designs," In Proc. 2016 CHI Conf. Hum. Factors Comput. Syst. (CHI '16), 2016, pp. 5494–5505.

[34] Michael Nebeling, "XDBrowser 2.0: Semi-Automatic Generation of Cross-Device Interfaces," In Proc. 2017 CHI Conf. Hum. Factors Comput. Syst. (CHI '17), 2017, pp. 4574–4584.

[35] "IP Webcam - Apps on Google Play," https://play.google.com/store/apps/details?id=com.pas.webcam&hl=en_US&gl=US.

[36] Yu Wen Jong, Pi Cheng Hsiu, Sheng Wei Cheng, and Tei Wei Kuo, "A semantics-aware design for mounting remote sensors on mobile systems," In Proc. 53rd Annu. Des. Autom. Conf. (DAC '16), 2016, pp. 1–6.

[37] Kaisen Lin, David Chu, James Mickens, Li Zhuang, Feng Zhao, and Jian Qiu, "Gibraltar: Exposing Hardware Devices to Web Pages Using {AJAX}," In 3rd USENIX Conf. Web Appl. Dev. (WebApps 12), 2012, pp. 75–87.