Weave: Scripting Cross-Device Wearable Interaction

Pei-Yu (Peggy) Chi [*]
Computer Science Division, UC Berkeley
peggychi@cs.berkeley.edu

Yang Li
Google Inc.
yangli@acm.org

ABSTRACT

We present Weave, a framework for developers to create cross-device wearable interaction by scripting. Weave provides a set of high-level APIs, based on JavaScript, for developers to easily distribute UI output and combine sensing events and user input across mobile and wearable devices. Weave allows developers to focus on their target interaction behaviors and manipulate devices regarding their capabilities and affordances, rather than low-level specifications. Weave also contributes an integrated authoring environment for developers to program and test cross-device behaviors, and when ready, deploy these behaviors to its runtime environment on users' ad-hoc network of devices. An evaluation of Weave with 12 participants on a range of tasks revealed that Weave significantly reduced the effort of developers for creating and iterating on cross-device interaction.

Author Keywords

Scripting; cross-device interaction; wearable computing; mobile computing, UI tools; gesture-based interaction.

INTRODUCTION

Mobile and wearable computing gadgets are flourishing. For example, smartphones have become an all-purpose computing device for our everyday activities; smartwatches provide glanceable information and instant access to a set of frequent tasks; and smart eyewears such as Glass [13] enable users to navigate private content and receive prompt assistance in their peripheral vision. Each of these devices offers a distinctive form factor, and input and output capabilities that are designed to seamlessly blend into users' existing artifacts and daily activities.

To allow users to fully leverage a multi-device ecosystem in their lives—a vision of ubiquitous computing [31]—it is important to combine the unique strengths of each device and to enable rich and fluid interaction behaviors across devices [6,23]. Prior work has investigated a variety of behaviors that can be formed by orchestrating a smartphone and a watch [4]. Existing commercial products have begun to support basic cross-device behaviors, such as browsing an image gallery on the phone with a watch remote [14] or navigating media on the TV with a phone [12].

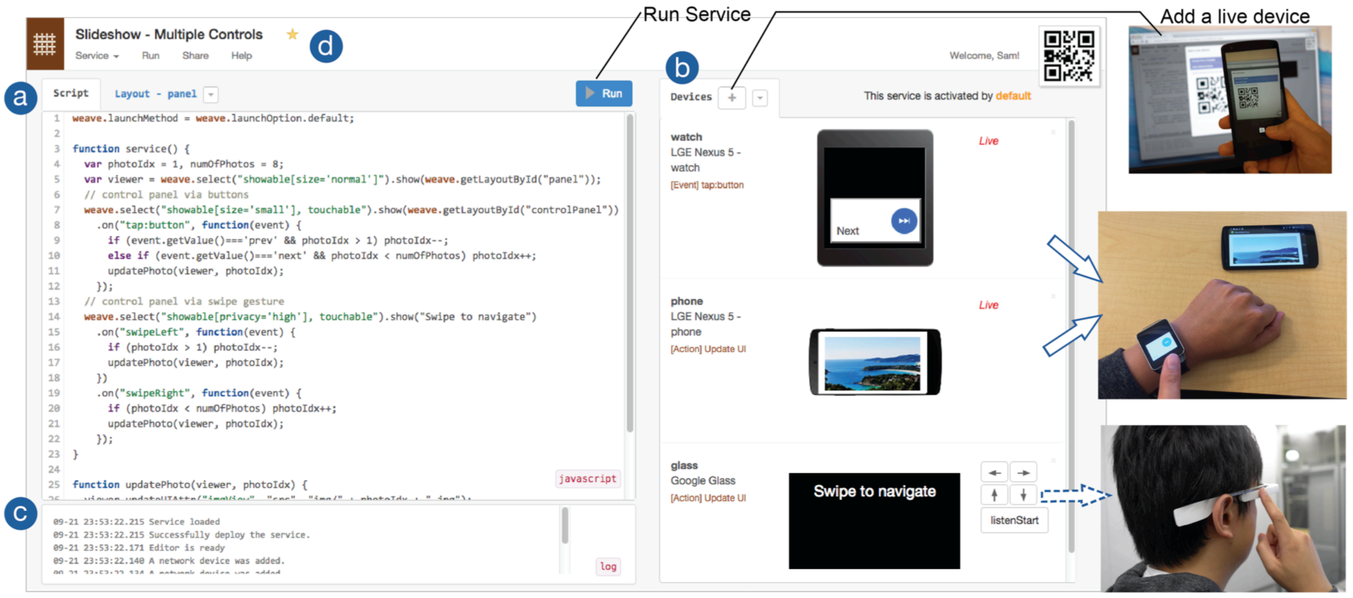
However, programming cross-device wearable interaction remains challenging. To create an interaction behavior that spans multiple wearable and mobile devices, developers have to design and implement based on the varying input and output resources and preferred interaction styles of each individual device. Developers also need to distribute user interfaces and synthesize user input and sensing events across devices. Realizing these tasks manually involves many low-level details such as dealing with device-specific constraints, communication and synchronization across devices, which is technically challenging and effort consuming. It distracts developers from concentrating on their interaction behaviors and results in a system that is often difficult to maintain and adapt to new types of devices or device combinations that might be available at runtime.

To address these issues, prior work has investigated tool support for creating cross-device interaction. In particular, Panelrama introduced mechanisms for automatically distributing a web interface across multiple displays based on the specified preferences of each UI segment and the screen real estates of target displays [32]. XDStudio provided a GUI builder for visually authoring a distributed user interface in a simulated or an on-device mode [25]. However, there is still a lack of high-level support for developers to easily handle interaction events that are synthesized from user input and sensed physical events across multiple devices, which is crucial for creating rich cross-device wearable interaction as shown previously [4]. Moreover, few have considered abstractions and mechanisms for accessing wearable devices regarding their affordances, such as glanceable and shakable natures. These properties are unique aspects of programming cross-device wearable interaction.

In this paper, we propose *Weave*, a framework that allows developers to easily create cross-device wearable behaviors by scripting. Weave provides a web-based development environment (see Figure 1) where developers can author interaction scripts and test them based on a set of simulated or real wearable devices anywhere and any time. Weave also provides a runtime environment that allows Weave scripts to run in users' ad-hoc network of wearable devices

CHI 2015, April 18–23, 2015, Seoul, Republic of Korea
ACM 978-1-4503-3145-6/15/04.
http://dx.doi.org/10.1145/2702123.2702451

* This work was done while the author was an intern at Google.



```javascript
weave.launchMethod = weave.launchOption.default;

function service() {
  var photoIdx = 1, numOfPhotos = 8;
  var viewer = weave.select("showable[size='normal']").show(weave.getLayoutById("panel"));
  // control panel via buttons
  weave.select("showable[size='small'], touchable").show(weave.getLayoutById("controlPanel"))
    .on("tap:button", function(event) {
      if (event.getValue()==='prev' && photoIdx > 1) photoIdx--;
      else if (event.getValue()==='next' && photoIdx < numOfPhotos) photoIdx++;
      updatePhoto(viewer, photoIdx);
    });
  // control panel via swipe gesture
  weave.select("showable[privacy='high'], touchable").show("Swipe to navigate")
    .on("swipeLeft", function(event) {
      if (photoIdx > 1) photoIdx--;
      updatePhoto(viewer, photoIdx);
    })
    .on("swipeRight", function(event) {
      if (photoIdx < numOfPhotos) photoIdx++;
      updatePhoto(viewer, photoIdx);
    });
}

function updatePhoto(viewer, photoIdx) {
```

```
09-21 23:53:22.215 Service loaded
09-21 23:53:22.215 Successfully deploy the service.
09-21 23:53:22.171 Editor is ready
09-21 23:53:22.140 A network device was added.
```

Figure 1. The Weave authoring environment. Developers can (a) script cross-device behaviors and specify associated UI layouts in the Edit Panel, (b) test these behaviors on emulators or live devices via the Test Panel, (c) debug the scripts by observing detailed output in the Log Panel, and (d) manage and share a script via the menu. In this figure, the Test panel contains two real devices, i.e., watch and phone, and one emulator device, i.e., glass. Alternatively, the glass emulator can be replaced with an actual Google Glass device.

and easily adapt to specific device configurations. Weave contributes a set of high-level abstractions and APIs that capture affordances of wearable devices and provide mechanisms for distributing output and combining sensing events and user input across multiple devices, using high-level clauses such as "select" and "combine". These APIs combined with Weave's integrated authoring and runtime support enable a new approach for programming event-based cross-wearable interactions, which allows developers to focus on target interaction behaviors rather than low-level implementation details.

RELATED WORK

Weave builds on prior work of cross-device interactions and tools for creating these behaviors. We review and discuss our relationship with the art in these areas.

Wearable & Cross-Device Interactions

Mobile and wearable devices equipped with lightweight sensors have introduced new possibilities for interaction modalities. Versatile commercial wrist-worn products and phones are able to track people's physical activities [9] and perform daily tasks such as automatic voicemail playback [1]. Eyewears enable users to blend digital content with the physical world [8], navigate content with eye blinks [20] or hand-to-face gestures [28], control remote objects [33], and track health status [19]. Wagner *et al.* studied how body-centric interaction techniques can be used to manipulate digital content in a multi-surface environment [30]. By locating the relative positions of hand-held devices to body joints, Chen *et al.* investigated techniques that allow users to retrieve virtual content by physical movement around the body, such as orienting devices [5].

To form rich behaviors based on limited but versatile I/O capabilities of each wearable device, prior work has investigated novel interactions beyond a single device. Studies have revealed that users tended to allocate a complex task across multiple devices based on the device form factors and functionalities [6,27]. Recently, Duet demonstrates a variety of intriguing behaviors, such as cross-device pinch gestures and finger knuckle and tip inference by fusing sensor input across a watch and a phone [4]. Existing commercial products have started to support cross-device behaviors such as using a watch as a remote to control content on a phone [14] or automatic task continuation when the user moves from a phone to a computer [1]. These scenarios inspired us to design high-level tool support for developers to design and create cross-device wearable interaction, which would otherwise be difficult to implement.

Prior work has extensively investigated cross-device interaction in a multi-display, multi-user environment [7,17,11]. In these situations, handheld devices are often used as a remote control to manipulate digital content on large displays [2,18]; physical distance (often referred as proxemics) and orientation relationships between devices are frequently used as cues in these scenarios [22,23]. We focus on a body-centric cross-device situation in which multiple wearable devices are intended to perform as an organic whole for the user, although Weave can be applied to addressing more general cross-device scenarios.

Tools for Creating Cross-Device Wearable Interactions

While cross-device wearable interaction is promising, it is extremely challenging to develop due to the complexity in combining multiple heterogeneous devices to form fluid interaction behaviors. To address this issue, prior work has developed several toolkits to simplify the distribution of UI content across devices. Yang and Wigdor developed Panelrama, a framework for distributing a web UI across multiple displays. Using linear optimization, Panelrama dynamically allocates each portion of the UI to a display based on its screen real estate and input modality [32]. Other approaches include designing high-level UI abstractions, e.g. [24,26], and runtime frameworks for distributing UIs to a dynamic environment, e.g., [10].

Sinha explored informal prototyping tools for iterative design of multidevice user interfaces, which are not aimed for generating functional systems [29]. Recently, Nebeling *et al.* developed XDStudio, a visual tool for interactively authoring cross-device interfaces [25]. With XDStudio, developers can design cross-device interfaces with simulated devices or directly on real devices. It also allows developers to specify how a UI should be allocated by authoring distribution profiles. Similar to XDStudio, Weave allows developers to test their scripts with simulated devices, real devices or a mix of them.

While prior systems provide tool support for distributing user content across displays, they are insufficient for creating complex cross-device wearable interaction in two ways. First, there is a lack of high-level support for event handling across devices especially for combining user input and sensing events from multiple devices. This is crucial for wearable interaction because a user action often involves a collaboration of multiple wearable devices that forms a compound event as shown previously [4]. We designed Weave to provide a set of high-level API mechanisms for easily synthesizing events across devices. Second, Weave allows developers to script based on abstractions of devices regarding their affordances and capabilities. These types of abstractions, which are not shown in previous work, greatly simplify the creation of wearable interaction and allow developers to focus on high-level target behaviors.

SCRIPTING CROSS-WEARABLE INTERACTION

We discuss how a developer creates cross-device wearable behaviors in Weave, which we refer to as a Weave service. A Weave service consists of an interaction script, based on JavaScript, and a collection of UI layout specifications in HTML. A Weave service runs across devices and stitches capabilities on these devices. To help us illustrate the use of Weave, assume a programmer, Sam, who wants to design and implement a "Smart Messenger" service that brings an incoming message to the user's attention and allows the user to easily play and respond the message using a combination of wearable devices.

Creating a Weave Service

To create a Weave service, Sam opens the Weave authoring environment by pointing a web browser to the Weave server (see Figure 1). This environment allows him to create a new service, edit an existing service, and later share a service such that users can discover and install it on their devices. The Weave authoring interface has three parts: On the left is the Edit panel where the developer can edit Weave scripts and describe UI layouts in HTML. On the right is the Test panel where the developer can specify a set of wearable devices, simulated or real, and test the Weave service on these devices. The Log panel at the bottom of the screen helps developers to debug their scripts.

Sam starts by describing the name of the service and how the service is supposed to be launched, e.g., in response to a notification or when the user manually invokes it. Here Sam specifies the service to be invoked by an incoming message (notification) (see Figure 2a). To show an incoming message on a device that is easily `glanceable` by the user and equipped with a speaker, Sam adds several criteria in Weave's `select` clause (see Figure 2b) and refers to the selected device by a variable `notifier`. Each device that is Weave-capable maintains a device profile that captures the device's I/O resources such that Weave can match these selection criteria at runtime. Sam also requires the device's input capability to be touch-sensitive, i.e., `touchable`, and detect two motion events, `shakable` and `rotatable`. These criteria will select a watch if it is available. In the following discussion, we use "watch" and `notifier` interchangeably.

Sam then specifies the content to show on the selected device in the *Layout* tab, which consists of a title message and two buttons (see Figure 3). The sender's name will be updated at runtime when the service is invoked (see Figure 2c). To show this UI on the `notifier` device, the developer simply uses `notifier.show(ui)` (see Figure 2d). Depending on what device is selected by Weave at runtime, the UI will be rendered by Weave differently to be consistent with device-specific UI guidelines.

To handle button touch events, Sam adds a callback function (see Figure 2e). Based on which button is tapped by the user, the service either plays the message or texts back the sender via the device that originates the button-tap event—`event.getDevice()`, i.e., the `notifier` device.

To playback the voice message automatically when the user intends to listen to the message privately by bringing a smartphone to the ear, Sam adds another behavior to the service. He selects a device that affords private playback and subscribes to the `listenStart` event (Figure 2f), which is a built-in sensing event fired by Weave's runtime environment when it detects a phone "listen" action triggered by the user. Similar to existing commercial products, we detect this activity based on the phone's proximity sensor and accelerometer.

While the message is playing on the phone, a text message will be shown on the original `notifier` device to prompt the user for available options: the user can shake the watch to replay the message on the phone, or rotate the watch to call back the sender. The developer adds two chained event

```
a)  1 weave.launchMethod = weave.launchOption.notification;
    2 function service(notif) {
b)  3   var notifier = weave.select(
    4     "showable[glanceability='high'], speakable,"
    5     + "touchable, shakable, rotatable");
    6   var ui = weave.getLayoutById("mainPanel");
c)  7   ui.updateUI("callerName", notif.caller.name);
d)  8   notifier.show(ui)
e)  9     .on("tap:button", function(event) {
   10       if (event.getValue() === "playMsg") {
   11         event.getDevice().play(notif.voiceMsg)
   12           .show(weave.getWidget("mediaPlayer"));
   13       } else if (event.getValue() === "text") {
   14         event.getDevice().startApp("Messenger",
   15           notif.caller.name);
   16       }
   17     });
f) 18   weave.select("speakable[privacy='high'], phoneable")
   19     .on("listenStart", function(event) {
   20       var phone = event.getDevice().play(notif.voiceMsg);
   21       notifier.show("Shake to reply / rotate to call back")
   22         .on("shake", function(event) {
   23           phone.play(notif.voiceMsg);
   24         }).on("rotate", function(event) {
   25           phone.call(notif.caller.num);
   26         });
   27     });
   28 }
```

Figure 2. The complete script of the Smart Messenger service.

```
<div id="mainPanel">
  <p>New voice message from
    <span id="callerName"></span>
  </p>
  <button value="playMsg">Play</button>
  <button value="text">Text back</button>
</div>
```

Figure 3. An example of a UI layout. This layout is referred by the script at Line 6 in Figure 2.

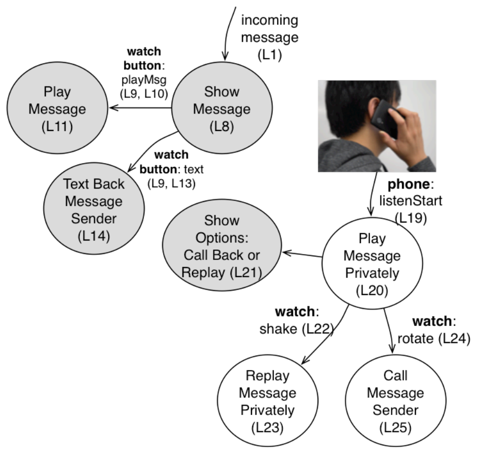

incoming message (L1)
Show Message (L8)
watch button: playMsg (L9, L10)
Play Message (L11)
watch button: text (L9, L13)
Text Back Message Sender (L14)
phone: listenStart (L19)
Play Message Privately (L20)
Show Options: Call Back or Replay (L21)
watch: shake (L22)
watch: rotate (L24)
Replay Message Privately (L23)
Call Message Sender (L25)

Figure 4. The interaction flow of the Smart Messenger. The shaded circles denote the watch's states and the plain circles for the phone's. The corresponding line of code is labeled.

handlers (Line 22 to 26) to respond to the motion events on the `notifier` device: `shake` and `rotate`. These options allow the user to continue the task with the watch while the user keeps the phone to the ear that is less accessible for further interaction. This example demonstrates that Weave allows developers to easily combine the strengths of multiple devices and distribute an interaction flow across devices to fulfill a task (see Figure 4).

Testing & Deploying a Weave Service

To verify if a service behaves as expected, Sam tests the service in Weave by running it on emulators, real devices, or a mix of them. Weave's authoring environment provides a collection of predefined emulators, which include typical wearable devices such as a watch, and several presets of common device combinations. These allow developers to test a service with minimal effort. Each emulator device allows a set of built-in events, such as `swipe` and `listenStart`, shown along the emulator in the *Test* panel (see the Glass emulator in Figure 1). Sam can tap on these events to trigger actions. He can also create new emulators that best characterize their target devices.

To add a real device to the test, Sam scans the QR code shown above the Test panel or in the dialog box via the + button (see Figure 1), using the Weave proxy running on a device. The screen of a registered device will be displayed in this panel, which is marked as *Live* in red. With actual devices, Sam can physically perform actions such as shaking the device or tapping the buttons. Any changes on a device's UI will be synced with its representation in the Test panel for the developer to monitor the output of multiple devices in real time. The *Log* panel at the bottom of the screen offers detailed information for debugging a service, including the event history and script outputs.

THE DESIGN OF THE WEAVE FRAMEWORK

We aim to provide a set of high-level abstractions for developers to easily program complex cross-device wearable interactions. A common pattern for modern UI development follows an object-oriented event-driven paradigm, where developers attach event handlers to a UI object (or a widget) that update the object accordingly when an event occurs. We are especially inspired by popular JavaScript library designs such as jQuery and d3 [3] that provide a high-level abstraction for programmers to manipulate UI components. One can easily select desired HTML elements to change their properties, e.g., `$(".target_class").css("color", "white")` changes the color of all the elements of `target_class` to white, or to handle user input events, e.g., the inline function of `$("button").click(function(){...})` will be invoked when any button is clicked by the user. Providing function calls based on high-level abstractions allows developers to focus on target behaviors rather than low-level operations, such as device specifications and network connection. Based on the unique challenges raised by programming cross-device wearable interaction, we designed Weave's scripting framework (see Table 1).

Selecting Abstract Devices

It is challenging for developers to cover various types of devices and their combinations that might occur at runtime. For example, some users may carry a smartwatch paired with a smartphone, while others may use an eyewear with a phone. Therefore, in addition to directly retrieving a device by its specific type, a framework should enable flexible selection of devices based on their high-level input and output capabilities so that a cross-device service can adapt to specific sets of devices that a user carries at runtime. As the earlier example shows, developers may choose to stream a message to a glanceable display, which at runtime could be a watch or an eyewear available on the network. With Weave, a developer selects target devices using the `select` clause with a selector of format "`CAPABILITY[PROPERTY='VALUE'],...`". Our framework returns a `WeaveSelection` object that includes one or more qualified devices to be operated. Developers can use multiple criteria at the same time to refine the selection. The following example selects devices with a small, glanceable display that detects rotation motion, which could be a smartwatch at runtime.

```
weave.select("showable[glanceability='high',
             size='small'], rotatable")
```

To select devices that can make phone calls, afford high privacy, and have a touch-sensitive display, developers may use the following criteria, with which Weave may find a smartphone and an eyewear at runtime.

```
weave.select("phoneable[privacy='high'], touchable")
```

In addition, our framework allows developers to easily select all the available devices using `weave.selectAll()`. Developers can also select devices of a specific type, e.g., `getDeviceByType("phone")`, or designed to be worn at specific body locations, e.g., `getDeviceByJoint("wrist")`, which is predefined in the device profiles. For fine-grained controls over device selection, developers can also directly fetch specific devices by their IDs, e.g., `getDeviceById("LGEphone")`. Finally, a negating operator allows the developers to exclude certain devices, such as `.not("speakable")` to select those without a speaker.

At runtime, Weave attempts to match these high-level criteria against devices that are available on the network. Each Weave-capable device runs on our runtime framework maintains a profile about its I/O capabilities, such as the display size or whether the device has a touchscreen, rotation sensor, or accelerometers. Analogous to conventional UI frameworks, each Weave device is treated as an interactive object in the context of a cross-device UI, which can be manipulated and registered to events.

Applying Actions to Devices

After specifying a selection, developers can apply actions to the device, such as providing output feedback. To show visual content on a device, `device.show(MESSAGE)` displays the message string in text, or developers can displays a UI layout defined as an HTML5 element in the Layout Panel using `device.show(weave.getLayoutById(LAYOUT_ID))`.

`Selecting Target Devices`	
`.select(selector)`	Return a `WeaveSelection` object that includes all the matched device(s) on a local network based on the selector string.
`Combining Multiple Devices`	
`.all()`	Return a `WeaveSelection` object that each device in the selection will behave the same.
`.combine()`	Return a `WeaveSelection` object that encapsulates the devices in the selection as one virtual device.
`Performing Output Actions`	
`.show(panel)`, `.show(string)`	Render a UI panel or a string on the device or the `WeaveSelection` object.
`.play(filepath)`	Play a media file of the file path on the device(s).
`.startApp(appname)`	Launch the specified native app.
`Handling Input Events`	
`.on(eventType, handler)`	Attach an event handler for one or more events to the device(s).

Table 1. Weave's core APIs.

To realize a UI layout on devices with different form factors and interaction styles, Weave takes into account the design guidelines of specific devices. It adopts the concepts of responsive web design that adapts the UI elements to different I/O capabilities in order to provide optimal viewing and interactive experience [21]. Our engine distributes the UI based on HTML tags (e.g. `div`, `span`, `p`, `img`, and `button`) and their attributes in a layout specification. For a UI layout that contains a text string (e.g. "Launch Pad Options") and three buttons ("Maps", "Email", and "Calendar"), on a phone with a high-resolution display, Weave shows all these components on the same pane and distributes the buttons vertically. On a watch with limited screen real estate, Weave displays the components in a grid view with each as an interactive card—a design metaphor for commercial smartwatches [15]. Similarly, on an eyewear they are represented as a list of swipeable cards that users can swipe through and tap to select [16]. Developers can update the UI shown on a device referred to by an element `id` using `device.updateUI(ID, NEW_CONTENT)` for the inner content or `device.updateUIAttr(ID, ATTR, NEW_ATTR_VALUE)` for a certain attribute. These functions are useful for providing incremental feedback.

To fully leverage the functionality of each device, a Weave service can invoke native apps on the device from the script, using `device.startApp(APP_NAME)`. A service can also make a phone call using `device.call(PHONE_NUMBER)` and play a media file using `device.play(MEDIA_FILE)`.

Attaching Event Callbacks to Devices

Based on what UI elements and sensor events are available on a selected device, developers can add a callback function to the device to handle user input. For example, developers can easily program a remote control service that allows the user to launch an application on a target device (e.g., a smartphone) by selecting the app from a `launchList` on a different device (e.g., a smartwatch) (see Figure 5).

```
weave.select("showable[size='small'], touchable")
  .show(weave.getLayoutById("launchList"))
  .on("tap:li", function(event) {
    weave.select("showable[size='normal']")
      .not(event.getDevice())
      .startApp(event.getValue());
  });
```

Figure 5. The script snippet allows the user to launch an app on a "normal"-sized device by selecting it from a list on a "small"-sized "touchable" device.

When the user taps on a list item that corresponds to an app name, Weave executes the callback function. The `event` argument passed into the function stores detailed event information, including the corresponding value retrieved by `event.getValue()`, and the device that triggers this event accessed via `event.getDevice()`. In this example, the callback function launches the app designated by `event.getValue()` on a device that has a normal-sized screen and is not the device where the event is triggered.

Weave provides other useful built-in events including `shake`, `rotateCW`, and `rotateCCW` inferred from motion sensors, `listenStart` and `listenEnd` if the user approaches the device to his ear, and common touch events such as `doubleTap`, `longPress`, and `swipe`. Note that our framework is designed to support device selection chaining and event propagation so that developers can attach the above operations in sequence, such as showing UI, adding several event callback functions, and applying other actions.

Coordinating Cross-Device Behaviors

Leveraging the strengths of multiple devices can form rich interaction behaviors. However, it is challenging to coordinate the behaviors of multiple devices. For instance, developers might want to display an urgent message to all the devices at the same time to maximally acquire the user's attention. For another instance, developers might want to use the current orientation of a smartwatch to indicate a different mode for a tap event on a smartphone touchscreen, e.g., detecting a knuckle or a fingertip tap [4]. Programming event callbacks and UI output for a single device is often straightforward. But handling these tasks for multiple devices can be extremely complex. A framework should assist developers in fulfilling these tasks.

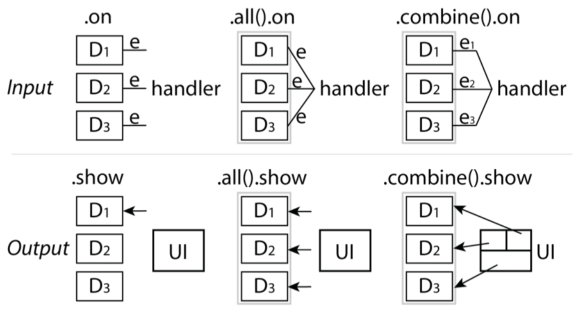


Figure 6. Weave coordinates multiple devices in three different modes (from left to right): default, `all`, and `combine` managing both input and output.

By default, when an event callback is attached to a selection (returned by `weave.select`) that includes multiple devices, Weave invokes the callback whenever one of these devices fires the event. Likewise, to execute an action on the selection, Weave selects a device in the selection that best matches the selection criteria, instead of performing the action on all the devices in the selection. On top of the default mode (which is similar to "any" device in a set), Weave provides two operators that can be applied to a selection for coordinating cross-device behaviors of these devices: `all` and `combine` (see Figure 6).

The `all` Operator

Developers can choose to enable multiple devices to behave simultaneously in the same way. One example is to broadcast the same message such that the message will be displayed on every device returned by `weave.select`. For coordinating input behaviors, the `all` mode requires a specific event to occur on all the devices in the selected set to invoke a callback function. For example, consider an interaction scenario that requires multiple users to physically bump their devices at the same time to form a user group. Realizing this scenario manually would require developers to collect the "shake" events from each device and compare their timestamps. To ease the development of this type of behaviors, Weave provides a high-level mechanism, i.e., the `all` operator, where developers can specify when and how devices should be synchronized. The code snippet in Figure 7 defines a callback function that is invoked when a shake event co-occurs on any two or more devices within a time range (500ms by default). In this snippet, each of the devices is required to have a display (`showable`), an accelerometer (`shakable`) for the shake event, and a speaker (`speakable`) for the audio output.

```
weave.select("showable, shakable, speakable")
  .all({minNumOfDevices: 2}) // at least 2 devices
  .show("Bump to join")
  .on("shake", function(event) {
    event.getDevices().show("Welcome to the group!")
      .play("audio/success.mp3");
  });
```

Figure 7. The `all` operator synchronizes the input and output of multiple devices and enables interaction scenarios such as bumping devices together to join a group.

The `combine` Operator

Weave introduces the `combine` operator that combines the input and output resources of multiple devices as an organic whole in which each device is allowed to perform a different part of a compound behavior. Via the `combine` operator, multiple devices within a selection essentially form a single virtual device that allows more diverse interaction behaviors.

Consider a cross-device slideshow scenario where a watch serves as a control panel for the user to go forward and backward through a list of images shown on a phone that can be viewed by others. To realize this scenario, developers apply the `combine` operator (Figure 8) to a selection of devices, e.g., a watch and a phone. Weave analyzes a UI layout that consists of an image viewer and two buttons and automatically distributes the buttons, which require minimum screen estate for "Previous" and "Next," to the watch and the image viewer, which requires higher resolution to be shown, to the phone. The distribution strategy is similar to Panelrama's [32] that primarily relies on preferred dimensions of each UI element. Note that an alternative script without using the `combine` operator is listed in Figure 1 (line 4-12) for comparison, which requires more variables to manage the device allocations for both the viewer and the control panel separately.

```
<div id="controlPanel">
 <button value="prev">Previous</button>
 <button value="next">Next</button>
 <img id="imageView" src="img/1.jpg"/>
</div>

var photoIdx = 1, numOfPhotos = 8;
weave.select("showable[privacy='low'], touchable")
  .combine().show(weave.getLayoutById("controlPanel"))
  .on("tap:button", function(event) {
    if (event.getValue()==="prev" && curPhoto > 1) {
      photoIdx --;
    } else if (event.getValue()==="next"
       && photoIdx < numOfPhotos ) {
      photoIdx ++;
    }
    event.getDevices().updateUIAttr("imageView",
      "src", "img/" + photoIdx + ".jpg");
  });
```

Figure 8. The `combine` operator distributes UI elements considering the screen estates and user interactions.

In addition to distributing a user interface, the `combine` operator allows developers to fuse different types of input events from multiple devices to create a new interaction mode. For example, Figure 9 shows a cross-device pinch gesture that invokes the callback function when a watch on the wrist detects a left swipe and the phone (assuming holding in hand) detects a right swipe at the same time.

```
weave.selectAll().combine()
  .on("swipeLeft[joint='wrist'],
     swipeRight[type='phone']", function(event) {
       event.getDevices().show("Pinch detected");
  });
```

Figure 9. The `combine` operator supports cross-device gestures as a new interaction mode.

Internally, Weave listens to element events of a compound (e.g., `swipeLeft` and `swipeRight`) on each device in the combination and triggers the callback when these events occur approximately at the same time. Weave maintains a global event pool to buffer and synthesize events from multiple devices. In general, `combine` creates an internal representation to manage the I/O of the combined devices. Weave encapsulates all these complexities for developers.

THE RUNTIME ARCHITECTURE

Weave employs a web-based architecture (see Figure 10) for creating and deploying services. Developers can access

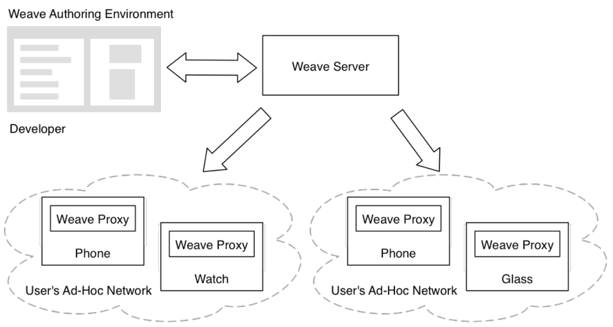


Figure 10. The architecture for creating Weave services.

Weave's authoring environment anywhere by pointing a web browser to the Weave server. Users can install a Weave service on their devices by accessing the Weave server using a Weave proxy.

A Weave proxy needs to be preinstalled and runs on each individual device. A Weave proxy serves two purposes. First, it provides tools for 1) generating a device profile that describes the capabilities of the device, 2) registering the device directly to the Test panel by scanning the QR code for developers to test a service on the device (see Figure 1), and 3) accessing the Weave server for searching and installing a service. A device profile is extracted automatically by analyzing the device resource (e.g. detecting available sensors), which can be refined by the owner using XMLs. Ideally the profile should come with the device, as it is specific and fixed to each type of device.

Second, a Weave proxy hosts a runtime environment for managing the ad-hoc network of wearable devices for a user and executing a Weave service on these devices (see Figure 11). To simplify the architectural design, Weave employs a client-server-based runtime architecture within each ad-hoc network. At any time, only one device on the

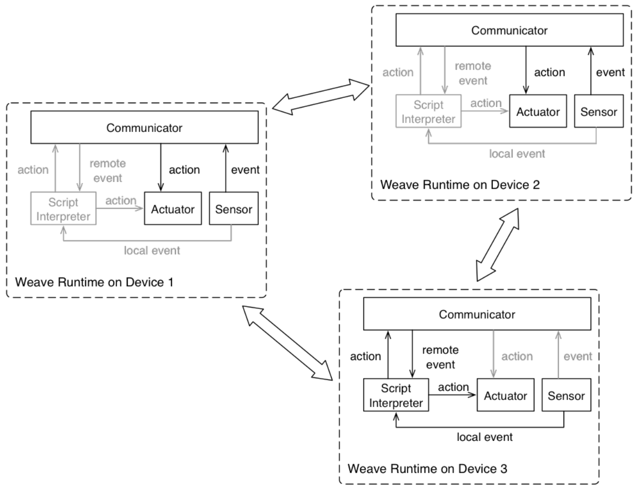


Figure 11. An illustration of Weave's runtime environment on a user's ad-hoc network. Here the network contains three devices, and Device 3 acts as the server for executing a script, and synthesizing events and distributing actions across devices.

network acts as the server that is responsible for executing the script and distributing the UI feedback and synthesizing events across devices. The runtime environment has four major components. The *Communicator* maintains connections with other devices on the ad-hoc network, and sends and receives events or actions between devices. The *Actuator* performs actions for updating the user interface on the device including playing audible or vibration feedback. The *Sensor* detects user input and sensor events on the device and sends the events to the *Script Interpreter*, which runs either locally on the device or remotely on another device on the network. The Script Interpreter executes a script and manages event callbacks and interaction states.

Any device can be added to the ad-hoc network or removed from it at runtime, as Weave promptly updates the device selection and status. When a server device leaves the network, another device on the network will succeed it as the server. This design allows flexible device configuration and stability in executing a cross-device service.

IMPLEMENTATION

The Weave web server is implemented based on Node.js to host the authoring UI. Weave's Script Interpreter executes scripts and HTML5-based UI layouts using a JavaScript engine. We have implemented and tested Weave proxies on three types of devices: the Nexus series of smartphones, Android Wear powered smartwatches such as Samsung Gear Live, and Google Glass.

The runtime server uses WebSocket and JSON to communicate with the Communicator running on each device. Note that since smartwatches such as Samsung Gear Live currently lack the web socket capability via Wi-Fi and only supports Bluetooth, they are paired with Android phones in order to connect to the server through the Wearable Data Layer API defined by the Google Play service. The Communicator on each device is responsible for redirecting traffic to a target device if the target device is not directly reachable from the server.

EVALUATION

To evaluate the usability of Weave's API framework design as well as the authoring environment as a whole, we conducted a user study with 12 participants. We hypothesized that Weave would allow developers to easily create a range of interaction behaviors across multiple mobile and wearable devices. We chose not to include a baseline condition (i.e., a control group) as our pilot studies showed that programming these interactions using the Android framework would take three hours at minimum.

Participants and Setups

We recruited 12 participants (2 females), aged between 20 and 48 years (Mean=28) from an IT company, including software engineers and interns. Participants were required to have moderate web programming knowledge and were selected randomly from volunteers via an internal study invitation. Participants had a varying programming background, ranged from 3 to 30 years (Mean=9 and SD=8). They had used two JavaScript tools on average, including jQuery, AngularJS, and Node.js. 9 out of 12 participants had mobile programming experience with Android phones, but only few had programmed wearable devices: two built basic Android Wear apps and one in Google Glass. Each participant was compensated with a $40 gift card for their participation.

The study was conducted in a quiet lab environment. A MacBook Pro running OS X and Google Chrome browser was connected to two 24-inch LCD displays of 1920x1200 pixel resolution. We provided an external mouse and keyboard. The set of mobile and wearable devices that are used in the experiment include two Android phones (Nexus 5 and Nexus), one Samsung Gear Live smartwatch that is powered by Android Wear, and one Google Glass eyewear.

Procedures and Tasks

To learn the framework, participants went through a tutorial and performed a practice task to create a single-device Launch Pad using Weave. Participants had the opportunities to learn different device selection criteria, specifying UI layouts in HTML, and adding callback functions to handle button taps. Participants also tested their service with emulators and real devices. The Weave API documentation was provided and available during the experiment. The learning phase was about 17 minutes on average. Participants were then asked to perform three programming tasks to create cross-device interactions using Weave.

Task 1: Modify the single-device Launch Pad created from the practice task to run a control panel on a glanceable device with three options (Maps, Email, and Calendar) and launch applications on a handheld device.

Task 2: Design a bump service that multiple users could join a group by physically shaking their devices at the same time. Provide both visual and audio feedback when successfully joined.

Task 3: Design a slideshow service that provides a panel on a small-size wearable for the user to browse a list of photos on a larger device—that is more viewable by others—by going forward or backward.

Participants could choose to test their scripts any time during the experiment. Finally, they were asked to answer a questionnaire, followed by a discussion with the experimenters. On average, the total duration of an entire user session lasted 80 minutes. We included some sample code from these tasks in Figures 5, 7 and 8.

EXPERIMENTAL RESULTS AND DISCUSSION

We discuss our observation about participants' coding strategies with Weave, their subjective feedback as well as opportunities for future work.

Code Analyses and Scripting Strategies

All the participants successfully scripted and tested the three tasks in the study. The average time of completing each task was 5, 11, and 18 minutes (SD=2, 3 and 6). The duration of Task 2 and 3 also included the time for participants to learn the `all` and `combine` operators. In general, participants were impressed by Weave's ease of use for development: P4 said, "*It was fast, easy to learn, and provided a way to unify cross platform devices with the same code. Would have taken a lot more time for me to do any of the tasks individually*." Compared to programming with the Android framework, all the participants suspected that it would have taken them at least three hours to multiple days for a single task, as they would have to work on the environment setup (P1), specific APIs (P8), and details such as network or message passing (P5).

The average number of lines of code within the "`service`" function was 6 ([*min*, *max*]=[5, 9], SD=1), 6 ([3, 9], SD=2), and 16 ([12, 22], SD=3) for each task, excluding the comments and code for system log printouts. The range varied for several reasons. Since Weave supports the chaining of functions, developers showed different styles of coding: some preferred to align operators in several lines, some contracted into one, and some created variables to manage. For a relatively complicated task such as Task 3, some participants chose to create an inner function to improve code readability. These behaviors showed that participants retained their JavaScript coding styles when scripting with Weave. P8 explained: "*I could interact with them (the devices) via a common interface that was easy to pick up with basic HTML, JS, and CSS (due to selector-like syntax) knowledge.*" P12 summarized, "*jQuery-like API is straightforward to most JavaScript programmers.*"

We observed that participants had a similar coding strategy: they started by selecting a set of devices and confirming the selected devices were as expected, then modified UI outputs and attached event listeners, and finally designed the callback logic. Such a coding strategy suggested that Weave supported a top-down and modular development in handling output, input, and interaction logic.

Feedback on Development

Participants were very positive about developing with Weave. They found it easy to learn the framework (4.6 on a 5-point Likert scale, SD=0.5), script a target behavior (4.6, SD=0.5), test a behavior (4.1, SD=0.9) and deploy it to real devices (4.8, SD=0.5). Participants commented: "*It was a fun experience. It was quite simple.*" (P11); "*Definitely easier than any other model I am aware of.*" (P10).

Framework Design

Participants found the abstraction and selection of Weave easy to manage: "*The CSS-like selection with Weave was pretty intuitive and made device selection very manageable. I also liked that it abstracted away the specific interface for each of the devices by allowing me to specify things that apply to (e.g.) any shakable device.*" (P8), and "*I like the concept of defining sets of devices by their use and semantic attributes (glanceable, private...).*" (P2). The results also showed that Weave's event-based structure supported interaction flow designs. P12 explained "*The API allows developers to implement cross-device interactions in functional programming paradigm is innovative.*"

However, participants desired more controls over distributing UI layouts across devices while using the `combine` operator. Previous mechanisms such as "panel" in Panelrama [32] can be useful in our framework for adding more developer controls to automate UI distribution. Some also wanted to see how Weave could embed an existing widget or web plugin in interfaces (P6, P8), which can be enabled in the future.

Service Deployment and Testing

Participants were surprised about the ease of use and the speed for deploying a service in real-time: "*Testing was great and easy. I definitely liked the immediate deploy of the code to the devices*" (P3); "*It was surprisingly easy to deploy the apps*" (P2). P1 liked the live testing and "*write once and run multiple*".

However, the debugging feature was limited with the current system. Several participants explicitly mentioned the need of highlighting syntax errors (P6, P8), code completion (P9), and writing test cases (P4). On average, the experimenters offered little help to participants for completing these tasks. The only type of help that was provided (less than two times on average for a task) was on fixing syntax errors, such as missing closing marks and incorrect built-in event names.

Opportunities

Participants expressed strong interests in scripting cross-device interactions with Weave (4.6, SD=0.7). Participants were also interested in potentially including more devices for content retrieval and manipulation, such as a desktop or a laptop (P2, P8), and a TV or other large monitors (P7, P9, P12). Some expected to integrate other services such as online photo collection or games (P9). All in all, the developer feedback showed that Weave provides a powerful tool as "*It significantly shortens the time required to prototype new ideas on wearable devices.*" (P12). We drew most of our examples and tasks from recent cross-device literature and commercial products, since our evaluation primarily focused on lowering the technical threshold, i.e., "lowering the floor". In the future, we intend to explore further on how our tool can raise the ceiling for creating more sophisticated cross-wearable behaviors.

We also identified the weakness of our framework and opportunities for future work. Weave currently does not support specification of relative spatial relations that have been used in prior work [22,32,5]. This limitation can be addressed by enabling developers or end-users to specify these spatial relations on devices [25] or enhancing with more sensors [5]. In addition, we are experimenting with features that allow developers to create custom cross-device events using programming by demonstration, and support multi-user scenarios by adding user identifiers, e.g., enabling `device.owner` as a built-in variable.

CONCLUSIONS

We present Weave, a framework for developers to create cross-device wearable interaction by scripting. We contributed a set of high-level APIs, abstractions and integrated tool support. Our evaluation with 12 participants indicated that Weave significantly reduced developer effort for creating complex interaction behaviors that involve UI distribution and event synthesizing across multiple wearable devices.

REFERENCES

1. Apple Inc. Apple – iOS 8 – Continuity. https://www.apple.com/ios/whats-new/continuity/ (2014).
2. Boring, S., Baur, D., Butz, A., Gustafson, S., and Baudisch, P. Touch projector: Mobile Interaction through Video. In *Proc. CHI '10*, ACM Press (2010).
3. Bostock, M., Ogievetsky, V., and Heer, J. D3: Data-Driven Documents. IEEE Transactions on Visualization and Computer Graphics 17, (2011).
4. Chen, X., Grossman, T., Wigdor, D.J., and Fitzmaurice, G. Duet: Exploring Joint Interactions on a Smart Phone and a Smart Watch. In *Proc. CHI '14*, ACM Press (2014).
5. Chen, X., Marquardt, N., Tang, A., Boring, S., and Greenberg, S. Extending a mobile device's interaction space through body-centric interaction. In *Proc. MobileHCI '12*, ACM Press (2012).
6. Dearman, D. and Pierce, J.S. "It's on my other computer!": Computing with Multiple Devices. In *Proc. CHI '08*, ACM Press (2008).
7. Elmqvist, N. Distributed user interfaces: State of the art. In *Distributed User Interfaces*. Springer London, 2011, 1–12.
8. Feiner, S. and Shamash, A. Hybrid user interfaces: breeding virtually bigger interfaces for physically smaller computers. In *Proc. UIST '91*, ACM Press (1991).
9. Fitbit Inc. Fitbix Flex. http://www.fitbit.com/flex (2014).
10. Frosini, L. and Paternò, F. User interface distribution in multi-device and multi-user environments with dynamically migrating engines. In *Proc. EICS 2014*, ACM Press (2014).
11. Gjerlufsen, T., Klokmose, C.N., Eagan, J., Pillias, C., and Beaudouin-Lafon, M. Shared substance: developing flexible multi-surface applications. In *Proc. CHI '11*, ACM Press (2011).
12. Google Inc. Chromecast. http://www.google.com/chrome/devices/chromecast/ (2014).
13. Google Inc. Google Glass. https://www.google.com/glass/start/ (2014).
14. Google Inc. Android Wear: The developer's perspective. https://www.google.com/events/io/io14videos/ (2014).
15. Google Inc. UI Patterns for Android Wear. https://developer.android.com/design/wear/ (2014).
16. Google Inc. Google Glass – Patterns. https://developers.google.com/glass/design/patterns (2014).
17. Hamilton, P. and Wigdor, D.J. Conductor: enabling and understanding cross-device interaction. In *Proc. CHI '14*, ACM Press (2014).
18. Hartmann, B., Beaudouin-Lafon, M., and Mackay, W.E. HydraScope: Creating Multi-Surface Meta-Applications Through View Synchronization and Input Multiplexing. In *Proc. PerDis '13*, ACM Press (2013).
19. Hernandez, J., Li, Y., Rehg, J.M., and Picard, R.W. BioGlass: Physiological Parameter Estimation Using a Head-mounted Wearable Device. In *Proc. MobiHealth '14* (2014).
20. Ishimaru, S., Kunze, K., Kise, K., Weppner, J., Dengel, A., Lukowicz, P., and Bulling, A In the blink of an eye – Combining Head Motion and Eye Blink Frequency for Activity Recognition with Google Glass. In *Proc. AH '14*, ACM Press (2014).
21. Marcotte, E. *Responsive Web Design*. A Book Apart (2011).
22. Marquardt, N., Diaz-Marino, R., Boring, S., and Greenberg, S. The proximity toolkit: Prototyping Proxemic Interactions in Ubiquitous Computing Ecologies. In *Proc. UIST '11*, ACM Press (2011).
23. Marquardt, N., Hinckley, K., and Greenberg, S. Cross-device interaction via micro-mobility and f-formations. In *Proc. UIST '12*, ACM Press (2012).
24. Mayer, S., Tschofen, A., Dey, A. K., Mattern, F. User Interfaces for Smart Things - A Generative Approach with Semantic Interaction Descriptions. ACM Transactions on Computer-Human Interaction (2014), 21, 2, article 12.
25. Nebeling, M., Mintsi, T., Husmann, M., and Norrie, M. Interactive development of cross-device user interfaces. In *Proc. CHI '14*, ACM Press (2014).
26. Paternò, F., Santoro, C., Spano, L.D. MARIA: A Universal Language for Service-Oriented Applications in Ubiquitous Environment. ACM Transactions on Computer-Human Interaction (2009), 16, 4, article 19.
27. Santosa, S. and Wigdor, D. A field study of multi-device workflows in distributed workspaces. In *Proc. UbiComp '13*, ACM Press (2013).
28. Serrano, M., Ens, B.M., and Irani, P.P. Exploring the use of hand-to-face input for interacting with head-worn displays. In *Proc. CHI '14*, ACM Press (2014).
29. Sinha, A. K. Informally Prototyping Multimodal, Multidevice User Interfaces. Ph.D. Dissertation. UC Berkeley (2003).
30. Wagner, J., Nancel, M., Gustafson, S.G., Huot, S., and Mackay, W.E. Body-centric design space for multi-surface interaction. In *Proc. CHI '13*, ACM Press (2013).
31. Weiser, M. and Brown, J.S. The coming age of calm technology. Beyond calculation: the next fifty years, Copernicus (1997).
32. Yang, J. and Wigdor, D. Panelrama: Enabling Easy Specification of Cross-Device Web Applications. In *Proc. CHI '14*, ACM Press (2014).
33. Zhang, B., Chen, Y., Tuna, C., Dave, A., Li, Y., and Lee, E., Hartmann, B. HOBS: Head Orientation-Based Selection in Physical Spaces. In *Proc. SUI '14* (2014).