XDBrowser: Challenges in Engineering a Next-Generation, Cross-Device Web Browser

Abstract
Recent research has focused on web developer toolkits for distributed multi-device user interfaces. We investigate a new solution, XDBrowser, where the web browser itself is aware of and able to use multiple devices in parallel. This paper discusses how XDBrowser's interaction and implementation techniques help overcome many challenges of BYOD-based interaction given its increasing ability to adapt existing web interfaces and browsers for cross-device use.

Author Keywords
multibrowsing; semi-automatic distribution; hybrid browser

Introduction
State-of-the-art web browsers have added support for keeping the browser history, bookmarks and settings in sync so that users can use multiple personal devices for browsing the web. This partly addresses the need for more seamless multi-device interaction identified in earlier studies on information work and web use [3, 9, 10, 14]. More recent studies [11, 15] find an increased need to better support parallel device usage so that users can flexibly distribute tasks between devices, taking into account both device capabilities and user preferences. However, there is no native browser support for using multiple devices in parallel. Instead, special cross-device development toolkits and modifications to existing web interface code are required [4, 7, 12, 17].
As part of the XDBrowser project, we are investigating how existing web browsers can be extended to support rich interactions and parallelism in cross-device use. A first prototype of XDBrowser [11] implemented new end-user customization tools for re-authoring existing web pages so that they can be distributed and synchronized between multiple devices. This prototype was then used to conduct a study on user-defined cross-device web page designs for a given set of five popular web applications. The study generated 144 cross-device designs that can be distilled down to seven core design patterns. While the first version of XDBrowser proved very useful for that study, we are currently working on a new version with two major improvements.

First, the existence of patterns suggests that part of the manual re-authoring process could be automated. Providing automated support for adaptation could be beneficial to users since users could browse new pages they have not visited before, without having to first customize them for cross-device use, and could instead choose from available patterns and switch the design depending on the task. The primary challenge then becomes to detect the desired pattern from only minimal user interaction and distribute existing web pages without prior modification by developers.

Second, while the chosen architecture was sufficient to enable our end-user customization study, there are technical limitations that we are currently addressing. We discuss the benefits and limitations of our browser-in-browser implementation technique which, rather than building on common browser extensions and plug-ins, overloads the browser with a full-screen browser interface that hides the host browser. It is compatible with a wide variety of devices and existing web browsers even if they do not support extensions, but still lacks some advantages of browser extensions that we aim to overcome with a new hybrid approach.

**Semi-Automatic Distribution Techniques**

Our end-user customization study using XDBrowser produced 144 desirable cross-device designs leading to seven distinct patterns. The full description of the study and results can be found in [11]. To illustrate the extensions we are designing for XDBrowser, let us focus on two patterns.

**Figure 1:** Six-step end-user customization in the first version of XDBrowser. Pattern-based semi-automatic distribution achieves the same result simply by double-tapping the inbox page element.

Consider the mail application in Figure 1 with the inbox and the reading pane distributed for remote-control from the phone. To distribute the elements in this way, the first version of XDBrowser required users to perform a series
of manual selections and operations to push selected elements between the devices. The goal of our new techniques is to reduce this effort to a single interaction. The main inspiration for our approach comes from modern browser support for double-tap to zoom on mobile devices to view selected portions of the page in more detail. Our idea is to allow users to double-tap the content they want zoomed and make use of connected devices to automatically distribute the page elements pushed out of the browser viewport when zooming the content on the current device.

Figure 2: Classification of DOM nodes for mail application

Figure 2 shows a breakdown of the main page elements relevant for two of the patterns we want to be able to activate, remote-control and overview+detail (Figure 3).

Since selecting a message in the inbox controls which message is shown in the reading pane, the remote-control pattern should become active when the user double-taps the Inbox element on the phone. As a result, the Inbox element should be kept on the phone and the other elements moved to the tablet (Figure 4(a)). Looking at how the Inbox element is constructed by nesting different types of HTML DOM nodes, from the node that received the double-tap event, we would need to traverse the DOM tree upwards until we find the node wrapping the Inbox element, i.e., the DIV with id “inbox” (Figure 2). This node is characterized by having an id attribute and containing a set of BUTTON and LI nodes with onclick event handlers. Once we have found this node, we can extract it and hide all other elements on the current device. On connected devices, we hide this node and show all other elements instead.

If the user double-taps the Message element, overview+detail should become active. As a result, the reading pane should be zoomed on the current device so that it fills the browser viewport (Figure 4(b)). In this case, the Message element is constructed from several nested DIV nodes, all of which again have an id attribute. Zooming any of them leads to the same result as zooming the “message” DIV directly.

Common web page segmentation techniques split the page into smaller blocks of content elements based on text and structure analysis, DOM hierarchy and layout information [1, 2, 6, 16]. These elements can then be extracted as a group. However, they typically require full analysis of the page content to do so, which can be computationally expensive, especially on less powerful devices. Based on insights from our study such as above, we are developing an interaction-based approach that does not require full segmentation and only involves relevant parts of the DOM tree. It constructs the DOM path to the invoked element, performs a classification into three types of elements—control, input, or other—by traversing the DOM hierarchy upwards, and activates the pattern depending on the type of element found by extracting relevant nodes and showing them on one device and hiding them on connected devices.
In our first evaluations with a set of 50 top-ranked sites by Alexa from 10 different genres, our simple classification proved sufficient to support semi-automatic distribution.

**Hybrid Approach to Cross-Device Browsing**
We developed two implementations of XDBrowser using different architectures. First, we developed an extension of the popular Chrome web browser, making it possible to run on both Desktop and Android devices, including tablets and smartphones. Note that on mobile devices we built on the Crosswalk web runtime¹ to embed the latest Chrome and add support for extensions which are not supported in Chrome for Android. The architecture is shown in Figure 5. XDBrowser connects multiple browser windows, either running on different devices or as multiple Chrome instances on the same device. The implementation is divided into a client-side **background script** executed once per browser window—it is used to activate patterns and maintain a WebSocket connection between multiple browser windows through the server; a **content script** executed for every page loaded into a tab—it is used to inject our DOM distribution and view state synchronization methods; and a Node.js server using Socket.IO for WebSocket communication. The Chrome API is used for DOM manipulation, Hammer.js for touch events, and Zoom.js to magnify web page elements that were extracted using our classification.

Our second implementation is “a browser within a browser”. The host browser, however, is not visible to the user since XDBrowser runs in fullscreen. This implementation has the advantage that iOS mobile devices and even Android Wear smartwatches on which Chrome is not available can be supported. Here, we use WIB² as the host browser instead. The client side of our second implementation uses responsive web design based on HTML5, CSS3 and jQuery to adapt to a wide range of devices including smartwatches and phones, tablets, desktops/laptops, and tabletops. Using only native web technologies allows it, in principle, to run on any web-enabled device with modern browser support. The server side is the same as the first implementation.

The two implementations have their pros and cons. The first is compatible with devices running Chrome and basically any web site, but requires installation of a browser plugin or special client. The second supports an even larger set of devices and any browser, but embeds web sites via iframes. Many top sites forbid iframe embedding and browsers prevent cross-site scripting, but a proxy server fixes this [5]. For sites that maintain a session, the user needs to login on each device, but a remote-control architecture [13] or shared virtual browser such as PhantomJS resolves this [8].

We are working on a hybrid approach that combines the best of these techniques (Figure 6). Using the common server side and parts of the Chrome API as the common interface, XDBrowser switches to the **browser-in-browser** approach if the host browser is not Chrome or the XDBrowser extension not installed. Note that the server can be embedded within the browser. We have also experimented with Peer.js rather than Node.js for peer-to-peer communication via WebRTC, which avoids the server after connection brokering and is especially useful for watch-based scenarios.

**Contribution to the CHI Workshop**
Participation in the Cross-Surface workshop provides an opportunity to discuss our XD-projects and exchange ideas on supporting BYOD-related interactions. We could give a demo of XDBrowser and highlight its potential for future studies in the wild. In particular, our implementation experience with different architectures will benefit the workshop.

---

¹ [https://crosswalk-project.org](https://crosswalk-project.org)
References


