Linköping University | Department of Computer and Information Science
Bachelor's thesis, 16 hp | BSc Computer Science
Spring term 2017 | LIU-IDA/LITH-EX-G--17/055--SE

Distributed User Interfaces for the Web

Anton Stevensson

Tutor, Anders Fröberg
Examinaner, Erik Berglund

LINKÖPINGS UNIVERSITET

Distributed User Interfaces for the Web

Anton Stevensson
Linköping University
Linköping, Sweden
antst190@student.liu.se

ABSTRACT

New ways of interaction between users and their computing devices has revealed that users are not only interested in sharing their data, but their user interfaces as well. This calls for an extension of the traditional notion of user interfaces, to a notion where user interfaces span over multiple devices: Distributed User Interfaces (DUIs). The purpose of this thesis is to identify challenges in developing Web based DUIs, and the result is a JavaScript library that aids the development of these types of interfaces. The development has been driven in an explorative fashion, to discover techniques that is appropriate to use, when moving ideas from existing DUI frameworks for the desktop, to the Web. Soundness of the implementation has been evaluated by measuring properties of the library source code that indicates maintainability and extensibility. The conclusion is that the library has the potential of answering the needs for future DUI development targeting the Web.

INTRODUCTION

Motivation

Distributed user interfaces (DUIs) are user interfaces where parts or the whole interface can be spread across multiple devices, screens and/or users. The emergence of the DUI concept is a consequence of the new way we interact with computers nowadays: today most people have a multitude of different interconnected devices like smartphones, tablets, laptops and desktop computers – mobile computing enables users to take their work with them, leave one computing environment for another, and continue their work from where they left off. Moreover, applications are getting more and more collaborative in nature due to their interconnectedness; they are used for engaging in collaborative tasks to reach a common goal. This new way of human-computer interaction has created the need for users to not only share their data, but their user interfaces as well.

It becomes more and more common, to ship applications with a Web browser user interface, rather than a traditional desktop user interface; it is therefore motivated to investigate how DUIs can be implemented in a Web setting. As of now, there is no built-in support in existing Web development tools for creation of this kind of interfaces – despite this, Web applications with features similar to those of DUIs already exists. In these DUI functionality is hardcoded into the application which make the code hard to reuse and forces programmers of new, similar applications to reinvent the wheel. Therefore, a more general approach that provides application programmers with tools for DUI development is desirable. A more general approach also opens opportunities to augment already existing single-user UI applications with DUI capabilities.

The development effort in creating the user interface is already major part of the development process. This became especially apparent in the shift from command line interfaces to graphical user interfaces (GUIs), and it is possible that the shift from GUIs to DUIs may result in a similar increase in complexity. Compared to traditional UI development DUIs introduces new challenges related to networking, shared UI state and variations in computing platforms etc. Without tools that aids development of DUIs the development experience is likely to get unnecessarily demanding.

Aim

The purpose of this thesis is to identify challenges in developing DUIs for Web applications and implement a client-side JavaScript library that illustrates how these challenges can be mitigated. The library shall have support to send portions of a Web page to other devices during runtime and hide complexity related to networking and UI state synchronization.

Research questions

The focus in this thesis will be to investigate appropriate methods for implementing a distribution mechanism that maintains usability and how to keep UI state consistent across multiple devices. More specifically:

- How can interactive functionality be preserved when Web page elements move between Web browsers?
- How can UI state of Web page elements visible in two or more browsers simultaneously be synchronized?

Delimitations

- In DUI related research, a common problem to study is how to automatically adapt UI components to new UI environments or how to improve usability by automatically choosing distribution target devices based on best-fit algorithms. This is beyond the scope of this thesis.
- The implemented framework is a proof-of-concept library that highlights certain techniques and tools. Efficiency and scalability is only taken into

consideration to the extent that it is feasible for a handful of connected devices and for Web page elements with a reasonable amount of complexity.
- Distribution of resources that requires browser extensions or plugins will not be supported by the library.

THEORY

Related work

Distributed user interfaces is a relatively young area of research, but neighboring research fields such as pervasive and ubiquitous computing, computer-supported collaborative work (CSCW), multi-device and multi-display environments etc. has implicitly touched upon aspects of DUIs before the term was frequently used. DUI research can be categorized into mainly two groups: those that focus on the DUI technology itself and those that makes use of DUIs in specific domains and for specific applications. The category most relevant for this thesis is those that treats DUI technology specifically, so only work in this domain will be presented here. In this category the aim is typically to create conceptual reference models or implementing software toolkits for DUI development. The reference models – such as the 4C model by Demeure et al [1] – defines the problem space, describes essential properties and gives a theoretical foundation that aids reasoning about DUIs. Research on DUI toolkits coincides with the goals of this work i.e. to provide a programming environment for DUI development.

One of the earliest work on general software for DUI development is that of migratory applications [2] where applications can move freely over the network, rather than being confined to a particular computer. Migratory applications are not able to utilize the full potential of DUIs however, as there is no support for partial migration of the user interface, which is a typical characteristic of DUIs. Melchior et al [3] presents a toolkit for peer-to-peer distribution of UI components, with no centralized server that contains the complete state of the user interfaces in the network. Distribution is done at the widget level and there is support for adaptation of these i.e. preserving usability when the presentation changes. Another toolkit with similar aims is that of Grolaux [4]. Both Grolaux and Melchior's solutions are platform independent and is implemented in the Mozart programming language due its network-transparent object model. Fröberg et al [5] proposes the Marve framework that has emerged from the development experience of a number of earlier projects that uses DUI technology. The approach is to build distribution functionality on top of an already existing GUI toolkit. They introduce a model for handling UI events in a distributed manner by controlling on which device event callback routines are executed. Marve has a reference implementation built on top of Java Swing, but their methodology is general and applicable to other GUI toolkits as well.

While the aforementioned frameworks targets the desktop, WebSplitter [6] brings DUI capabilities to Web applications, with support for splitting Web pages into personalized partial views that can be spread over various devices and users. Similarly, the Panelrama framework [7] introduces a XML element called panel, in which groups of HTML elements can be collected and shared between devices. Firmenich et al [8] uses Web augmentation as a tool for DUI development i.e. the use of browser extensions to execute client-side scripts.

Web applications with distributed UIs are also related to research done in the field of collaborative browsing (co-browsing) [9] – in co-browsing several users engage in a multi-device joint browsing session of the same Web page. Even though the goals of co-browsing and Web DUIs do not overlap completely, techniques and tools used for implementation are very similar. This is also the case for browser session migration [10][11] where the complete state of a Web application can be saved and restored later.

DUI Terms

Static and dynamic distribution

UI distribution can take place in different points in time – static distribution refers to distribution that occurs during design time i.e. development, compilation or start-up time. This is the least flexible type of distribution because it gives no ability to change the setup of the UI once the application has started. Dynamic distribution, on the other hand, means that distribution can happen during runtime, and as a consequence, the potential of applications where the user can initiate UI distribution.

Distribution granularity

In principle, any part of a UI can be transferred to other devices all the way down to the pixel level. Pixels in themselves do not carry any semantics and user interfaces often contain chunks of logically related entities. Therefore, a more viable alternative is to base distribution on the widgets of the local GUI toolkit, which by far is the most common approach in earlier work.

Component Singularity

Fröberg et al [5] introduces the notion of *component singularity*. In a distributed UI, components are visible at zero or more devices. *Atomic presentation* refers to components that are visible at a single device. *Mirrored presentation* is when a component is visible at two or more devices with shared UI state. *Cloned presentation* is when a component is placed on two or more devices, but the state of the component is unique to the device it resides on.

METHOD

Software that treats at least some aspect of the goals of this thesis already exists, so the design of the JavaScript library will take influence from these. The contribution of this thesis will not be to discover new underlying design principles for DUI applications, since this is already well covered in earlier work. It will instead be to apply concepts

from existing DUI toolkits for the desktop to a Web environment.

The development methodology will be a combination of iterative and experimental development – experimental in order to identify new techniques and concepts; iterative because acquired knowledge during the experimental phase often leads to the need of design revisions, when more feasible and effective solutions are discovered. Since iterative development leads to frequent changes in the code, it is important that the code is maintainable and easy to change. Furthermore, the library will lack some features present in other DUI frameworks, and it is possible that such features will be added to the library in the future, which further motivates the need of maintainable code. There have been substantial efforts in creating quantitative measures of code quality by the computer science community, and one of the results have been software metrics, which are measures to what degree software holds certain properties. Some common metrics are bugs per line of code, dependence between software modules (coupling), to which degree related functionality is grouped together (cohesion), the number of independent paths through the source code (cyclomatic complexity) etc. It is also common to combine different metrics, to measure certain qualities of the code such as maintainability, e.g. the maintainability index presented in [12].

To convince the reader that the results are valid and reliable, two case studies on two different kind of widgets, and a comparison with similar systems will be presented.

RESULTS

This chapter will present an overview of the features of the JavaScript library this worked has led to, two case studies that shows the library in action, a comparison to similar systems, and an evaluation of the source code.

Overview and features

Basic functionality

The API of the library is exposed through a global JavaScript function object – called *Distribution Manager* (shortened *dm* in the source code) – that attaches itself to the window object of a Web page. Distribution is done at the DOM/HTML element level and uses a wrapper – a *Distribution Object* – for these that DUI operations are issued upon. The Distribution Manager object are used in two ways:

- dm(<selector>)
- dm.<member_function>

The first way takes a reference or an identifier of a DOM node and returns a Distribution Object that exposes DUI functionality of that particular node e.g. operations for sending the node to another device. The second way reveals functions that acts on the DUI as a whole such as getting a list of connected users.

A device is added to the DUI network simply by the inclusion of a Distribution Manager object – each device has its own instance. The Distribution Manager automatically connects to a proxy server, that acts as an intermediate between browsers and relays unaltered metadata of DOM nodes. The proxy does not host any Web pages or keeps any kind of state of the user interfaces in the network. The library uses WebSocket [13] technology for asynchronous communication between browsers.

Distribution is initiated with one of the distribution primitives:

- distrObj.transfer(device, options)
- distrObj.mirror(device, options)

The *transfer* method is either move or a copy operation, which is specified by the *options* parameter. It copies or moves a DOM node from the call site to the device specified by the first argument. The *mirror* method is as the name suggests analogous to *mirrored presentation* – here, the options argument is used to specify whether the Distribution Manager shall perform synchronization of DOM trees automatically, or if the programmer wants to do this manually. The options argument can be used in both methods for specifying if the node should preserve placement constraints of the sending browser, which may be useful if the Web page of the sending and receiving browser is similar. User-controlled distribution is achieved by invoking above distribution operations in reaction to user triggered DOM events.

When a distribution operation is performed, metadata of the node subject for distribution is assembled by the Distribution Manager and sent to receiving devices via the proxy. At the receiving device, a programmer provided callback executes, which is registered with:

- dm.onReceive(function(node) { … })

The arriving DOM node is passed as an argument to the callback routine and lets the programmer insert the node to the Web page. As the library has no automatic mechanism for adding new nodes to browsers in a context-sensitive way, the current implementation puts this responsibility on the user of the framework. If the user does not specify a callback, the default behavior is to append the node to the body of the Web page.

In order to choose distribution target devices, a list of connected devices can be collected with:

- dm.users()

Programmers can react to users connecting and disconnecting by attaching a callback with:

- dm.userChange(function(user_list){..})

State and serialization of DOM nodes

Formalizing a way of representing UI components when they move from one device to another is the basis for making UI distribution possible. UI components usually have state that changes during the execution of an application, and this must be captured in a format that can be sent to other devices over the network, so they can be interpreted and reconstructed at receiving devices. For DUI toolkits that aims for platform independence, the transfer representation must be in the form of high-level UI description language, and use a middleware that interfaces the local GUI toolkit. The DUI library in this thesis targets Web browsers exclusively so such abstractions are not necessary – the semantics and structure of a Web page is described in terms of HTML. Not all aspects of a Web page can be expressed with HTML markup alone – dynamic behavior is obtained with client-side JavaScript code, but for simple Web page elements HTML is close to a complete description. The DOM API has a variety of ways to add HTML to a Web page dynamically during runtime, which makes it relatively straightforward to implement DOM node distribution. For DUI operations that preserves atomic presentation it only entails sending a snapshot of the nodes HTML markup at the time of distribution, and use functions in the DOM API to insert them in receiving browsers. The Distribution Manager uses APIs for DOM parsing and serialization [14] for this purpose – it is still experimental technology, but has support by all major Web browsers.

Even though transfer operations can be implemented by relatively simple means, mirroring operations requires additional implementation details. When a node is mirrored, a snapshot of the markup at distribution time is no longer sufficient, as state changes in one browser must result in the same changes in others. Hence, mirroring requires a mechanism that keeps the markup of mirrored nodes consistent across all browsers they reside on. For detection of markup changes, the Distribution Manager uses the DOM interface MutationObserver [15], that detects changes in DOM trees i.e. attribute changes, additions and removals of nodes. When an observer detects changes on a distributed node, information about the change is sent to affected browsers, where the local Distribution Manager updates the node accordingly. The information that is sent is either the name and value of a changed attribute, an identifier of a removed node or the markup and relative position of an added node. Even though this might seem meaningful only for mirrored nodes, the Distribution Manager uses observers on nodes that has been moved as well, which will be justified and explained in the next chapter.

For DOM nodes with JavaScript induced behavior, the use of MutationObservers is sometimes inappropriate, as it can lead to inconsistencies in the markup of a node and JavaScript engine state. Highly interactive widgets often rely its behavior on variables in JavaScript code, and just changing the markup is likely to lead to faulty behavior – as an example, imagine a group of radio buttons; they allow the user to choose from one (and only one) of a set of predefined options. When a user clicks on one of them, a script is executed that deselects the currently chosen option and marks a new one. Now suppose an attribute change notification arrives from the distribution proxy, and the Distribution Manager deselects the currently marked option and selects another one. If the click event script deselects based on the value of a JavaScript variable, that value is no longer valid. The next time the click event script is executed, it will deselect an already deselected option, select a new one, and there may now be two selected options – clearly this is not the intended behavior. For situations like this, the Distribution Manager lets the programmer handle state changes manually. This is supported through modification of a Distribution Object state property as follows:

- distrObj.setState(state)

When the method above is issued the Distribution Manager notifies other browsers that a state change has occurred, and executes a callback function registered with:

- distrObj.onChange(callback)

The state property can be any JavaScript object with the restrictions that its members are simple data types i.e. numbers, strings or Booleans. The callback that is executed in response to state changes on other devices takes appropriate actions based on the information of the updated state property – in the radio button example that would be to execute the click event script.

Event handling

Just sharing markup to implement UI distribution is not enough, however, since it does not preserve capabilities for user interaction to the full degree. The usual way for developers to react to user interaction in UI development is through event-driven programming. In an event-driven UI, the application listens for events corresponding to certain user actions, and triggers a callback/event-handler function when the event occurs. In a Web application an event-listener structure may be set up with HTML, but usually this is done in JavaScript code – thus, in order to preserve UI events more data than just HTML must be shared in order for distribution to function correctly. Consider as an example a login form: it most likely has an event handler that is executed when the user clicks the login button. When the click occurs, login credentials are sent to the Web server and then redirects to a page for logged in users. If the login form were to be sent to another browser without the event handler, clicking on the login button will have no effect. In other words, events and their callbacks must stay intact when moving between devices. Unfortunately, event handler code cannot easily be shared during runtime, unless unsafe JavaScript language constructs are used. The workaround is either to make sure that event handling code exists at compile time at potential target devices or to let

event handler code stay stationary such as described by Fröberg et al [5].

The standard way to register event handlers in Web development is through the DOM function *addEventListener()* – a Distribution Object has a function with the same name:

- distrObj.addEventListener(type, callback)

It works in a similar manner as the DOM equivalent, but with a few differences. The idea is to use this function instead of the DOM version in order to preserve event handling capabilities when moving nodes to other devices. The event handler code stays on the device where the function was invoked – but the event can be triggered by user interaction in other browsers as well. Even if the event handler code stays stationary, it can be used to emulate locally executed event handlers in other browsers quite effectively. When a node is distributed, regardless of whether it is mirrored or moved, a MutationObserver is attached to track changes of the node. In the case of a move operation, the node is removed from the DOM hierarchy and thus invisible to the user, but the Distribution Manager keeps a reference of it in memory. The reason for this is that event handlers often make changes to the event target, and if a node is moved and ceases to exist the event handler will not work any longer – the solution is instead to operate on the local copy kept by the Distribution Manager. Changes to the copy will be detected by the MutationObserver and these changes will be sent to other devices. This kind of event handling structure also serves an additional purpose other than emulating locally executed event handlers: certain event handlers are only meaningful in their initial context of a particular device, and are therefore inappropriate to distribute.

The solution of letting event handlers stay on the device where they were registered has a serious limitation however – once the browser where the event handler is located disconnects, other devices will lose their event handling capabilities. To solve this problem the library lets the programmer attach initialization scripts that is executed when a node arrives at a new device:

- distrObj.addInitScript(function_identifier)

The programmer passes the name of a function that must exist at potential distribution target devices. The typical use case for this function is to reattach event handlers that was lost in the distribution. Another is to set up logic for state changes with the *setState()* and *onChange()* functions.

Attributes and properties

HTML elements has a list of attributes with additional information such as its id, type, styling etc. The DOM representation of a HTML element has JavaScript properties that corresponds to these. There is not a one-to-one correspondence between them however, and their values are not always reflected. Input fields for example have a DOM property *value* that holds the current text of the input field. The HTML attribute with the same name contains the initial text of the input, but does not change as the user types.

Since the Distribution Manager distributes based on HTML, information is lost if JavaScript properties is not taken into regard – in the input field example, user input will not be preserved when distributed. Moreover, MutationObservers do not detect property changes unless they result in attribute changes as well.

To work around this problem, the Distribution Manager attempts to synchronize attributes and properties. Note that this must be done recursively for all descendants of a distribution target. The synchronization takes place at the time of distribution initiation. In the case of mirrored nodes, an event-listener structure for events associated with property changes is set up. These event listeners updates the HTML attribute of a changed property, which in turn results in a DOM mutation detected by the MutationObserver. Usually there is no need to attach event-listeners on all descendant of the distribution target, since events bubbles up in the DOM hierarchy. In the input field example, an event-listener is set to listen to the *change* event. When the event occurs, the event-handler will call the DOM *setAttribute()* method to update the *value* attribute.

Case studies and evaluation

Here a couple of distribution scenarios will be presented based on two case studies of two different kind of widgets.

Case study #1: Calculator widget

A calculator is divisible in the sense that it consists of a display and a keypad – it can be distributed either as a whole or partially distributed on either the display or the keypad. If distributed partially, say on the keypad, the intention is most likely that calculator should work as before but now on two different devices. In figure 1 we can see the keypad of a calculator that has been moved from user A's computer to user B's. In figure 2 we see the code of a calculator with primitive functionality. An event listener has been attached to all buttons of the keypad which updates the display based on which button that was clicked. The event handlers will stay in browser A even though the keypad has been moved to browser B (where clicks by the user will occur). Before the keypad was transferred the Distribution Manager saved a local reference of all the buttons which will be attached to the target property of the event object passed as an argument to the event handlers.

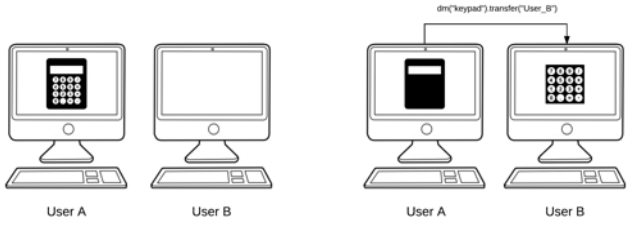

Figure 1. Partial distribution of a calculator widget.

```
<div id="display" class="screen"></div>
<div id="keys" class="keys">
 <button>7</button>
 <button>8</button>
 <button>9</button>
 <button class="operator">+</button>
  ...

 <button>0</button>
</div>
<script>
  var display = document.getElementById("display");
  var keys=document.getElementById("keys").children;
  keys.forEach(function(key) {
   dm(key).addEventListener("click", function(e){
      var input = display.innerHTML;
      var btnVal = e.target.innerHTML;
      if (btnVal == '=')
        display.innerHTML = eval(input);
      else
        display.innerHTML += btnVal;
     });
  });
</script>
```

Figure 2. HTML and JavaScript code for the calculator.

It is also possible to move and mirror the whole calculator (as opposed to only the keypad) with same code. From the Distribution Managers point of view there is little distinction between a move and mirror operation – in both cases a MutationObserver is attached to the distributed node, but in the case of a move operation the node becomes invisible to the user and only exists in memory. So let us say that the whole calculator has been either moved from browser A to B or are mirrored on both (the event handlers will still execute only in browser A). The following things will happen when a click event occurs in browser B:

1. An event object in browser B will be serialized and sent to browser A via the proxy.
2. The event handler is executed in browser A and the display is updated.
3. The MutationObserver will detect the change on the display and send mutation data to browser B.
4. The Distribution Manager will update the display in browser B based on the mutation information.

If a click event occurs in browser A only steps 3 and 4 will be performed. In the current implementation, cloning the calculator with the code in the figure would not be possible as this would require the Distribution Manager in browser A to have two distinct calculator copies – the local instance and another that browser B can base display updates from.

For the calculator widget there is no major differences in the code in order to make it distributable compared to non-distributable – the only difference is that the Distribution Manager function for handling events is used instead of the DOM *addEventListener()* function.

Case study#2 Color picker widget

Even if DOM tree synchronization works well in many cases, it is not guaranteed to always do so. When event handlers stay on the device they were attached, DOM tree synchronization always works – the reason for this is that changes to a node only is made in one browser and other browsers use it as a reference for markup updates. The complete JavaScript state of a node is contained in one browser and there can be no inconsistencies with HTML markup. When using initialization scripts to reattach event-handlers lost in distribution this is no longer guaranteed, and it is then more appropriate to let the application programmer handle the state of distributed nodes. Moreover, third party widget libraries are often used in Web development and these generally use their own event structure on top of native DOM events.

To illustrate how state changes of a mirrored DOM node can be handled manually, a color picker widget will be used. A color picker is a GUI widget that makes it easier to generate color codes by dragging the cursor inside a color map.

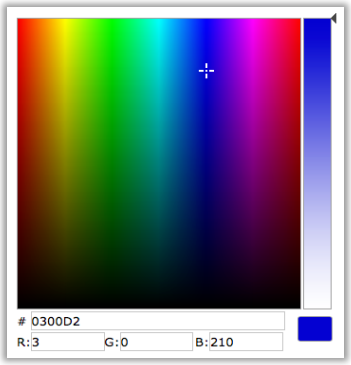

Figure 3. A color picker widget.

```
function colorPickerInit(distrObj) {
    $("#colorPicker").jqxColorPicker({ width : 350, height : 350 });
    $("#colorPicker").jqxColorPicker("setColor", distrObj.getState());
    $("#colorPicker").bind("colorchange", function(event) {
        distrObj.setState(event.args.color.hex);
    });
    dm("colorPicker").onChange(function(distrObj) {
        $("#colorPicker").jqxColorPicker("setColor",
                                         distrObj.getState());
    });
}
```

Figure 4. Initialization script for the color picker.

The color picker in this example is from jQWidgets[1], a widget library implemented in jQuery[2]. It has events that is triggered when a new color is chosen with the cursor, and has methods for setting the color programmatically. The programmer will handle color picker state by setting the state property of the DistributionObject to match the current color hex value. An initialization script for the color picker is shown in figure 4. This function must exist at receiving devices, and before distribution the sending device must call `dm("colorPicker").addInitScript("colorPickerInit")` in order for the *colorPickerInit* function to be executed when

[1] http://www.jqwidgets.com

[2] https://jquery.com

it arrives to the new device. The *colorPickerInit* function does the following:

1. Creates a colorPicker from an arbitrary HTML div element (in this case the distributed color picker div) with a constructor like syntax where width and height is specified.
2. Initializes the color to the one specified by the DistributionObject's state variable.
3. A callback is registered to fire in the event of a color change. The callback will modify the state variable of the Distribution Object accordingly.
4. A Distribution Manager listener is attached to listen for color changes on other devices. When that happens the listener callback will update the color programmatically.

When using this approach the Distribution Manager will not track changes in the DOM tree of the color picker. Changes to HTML markup is performed by the jQWidget library.

Comparative analysis

To make it more clear how DUIs in general, and the Distribution Manager library in particular, differs from similar technologies, a comparison between these is showed below.

	VNC	Migratory Applications/Web session migration	Distribution Manager API	EBL/TK
Control	External	External	Application	Application
Reproduction	No	No	Yes	Yes
Granularity	Full UI	Full application	Widget level	Widget level
Adaptation	No	No	No	Yes
Multi-user	No	No	Yes	Yes

The comparison is based on the following dimensions:

1. Control – if the application itself is distribution aware and controls the distribution, or if it is provided by a mechanism outside the application.
2. Reproduction – the ability for the UI, or parts of the UI, to be present at several devices concurrently.
3. Granularity – if the whole UI, application, or parts of the UI is distributed.
4. Adaptation – refers to the ability to adapt widgets (or the whole UI) by choosing the best presentation based on the situation and/or device.
5. Multi-user – if concurrent use by multiple users are supported.

The EBL/TK toolkit is the toolkit by Grolaux [4] mentioned in the theory chapter, and the dimensions above are taken from a comparison Grolaux evaluated EBL/TK with. VNC is a desktop sharing system, based on a client-server architecture. It makes it possible to remotely control another computer, by sending mouse and keyboard events, and get graphical updates as a response. As seen in the table, the EBL/TK toolkit and the Distribution Manager API are the systems that gives the greatest degree of freedom of what can be distributed. The major difference between the Distribution Manager API and EBL/TK, is the lack of adaptation in the former. Migratory UIs is a special case of DUIs able to distribute on the widget level.

Code analysis

To evaluate maintainability of the Distribution Manager code, a static code analysis tool has been used [16]. The tool uses a variation of the Maintainability Index, first presented in [12], which is included in Microsoft Visual Studio. It is calculated with the formula:

```
Maintainability Index =
  MAX(0, (171 - 5.2 * ln(Halstead Volume)
          - 0.23 * Cyclomatic Complexity
          - 16.2 * ln(Lines of Code)
        ) * 100 / 171)
```

A value between 0-9 indicates serious maintainability issues, values between 10-19 moderate issues, and values between 20-100 indicates that the maintainability is at an acceptable level. The Halstead Volume in the formula is calculated based on the numbers of operators and operands. The cyclomatic complexity is the number of the distinct paths through the code. Any value above 20 is a good indicator that the code is maintainable – the Distribution Manager get a score around 68. As a comparison the jQuery library get a score of 76.33, but even though the scores are close, one must take into regard that the jQuery code base is more than 10 times as large as the Distribution Manager API. The value tends to decrease when the number of lines of code increase, so one cannot draw the conclusion that the maintainability is at the same level.

For the cyclomatic complexity metric, values ranging from 1-10 are considered simple and easy to understand, values between 10-20 are considered complex, but may still be understandable. Higher values are generally considered to be very difficult to both grasp and maintain. The average cyclomatic complexity per-function in the Distribution Manager library is around 2 and the highest is 13 distinct paths through a function. The low average is due to fact that the library contains a large set of simple helper functions, which decreases the average quite substantially.

As mentioned in the delimitation section, the purpose of the Distribution Manager is not to be a full-fledged DUI library, but rather a proof-of-concept library to verify that certain models and concepts has a practical potential. The library has a bit over 500 lines of code, whereas the

EBL/TK toolkit as an example has well over 5000, which illustrates this rather clearly.

The code metrics above do not concern the applications used for testing, or the code for the proxy server. The proxy server has only been a minor part of the development effort and contains around 70 lines of Python code. The main application that was used for testing contains around 300 lines of HTML + 250 lines of JavaScript.

DISCUSSION

Results

An interesting use case of DUI libraries for the Web, is to add distribution capabilities on top of popular Web sites, where distribution intent is not coded into the application from the start. This is covered in great detail by the work of Firmenich et al [8], and their solution allows to create DUIs where users share Web page elements from a dynamic set of Web sites – for example, a user browsing *www.google.com* can send his or hers search result to someone browsing *www.wikipedia.org*. Their solution is quite different from mine, in that it goes well beyond just the sharing of DOM nodes – their framework has a rich set of facilities that makes collaborative cross-site Web browsing easier, e.g. data collectors, automatic form-filling and URL navigation, which my library lacks. They use Web augmentation – i.e. browser extensions that modifies a Web page – to add distribution functionality to distribution unaware Web sites e.g. Gmail or Wikipedia. In theory, it would indeed be possible to use the Distribution Manager API in combination with Web augmenting, however, there are very few circumstances where it would be very useful. Sharing static representations of Web page elements would work out-of-the-box with the transfer or mirror operations, but the usability of the element will not be preserved, since this requires the use of additional API functions. In other words, there are no neat way to distribute an arbitrary Web page element with a single command, as this requires per element-specific code – an interesting question is if it would be possible to implement a library that can distribute arbitrary DOM nodes without the need for specialized code. Unfortunately, this idea already falls with security issues regarding sending event listener code to remote Web browsers. Running remote code with local resources is almost always a bad idea, not only in Web development, as this in theory imposes no limitation on the harm a potential attacker could do. To implement distribution that automatically preserves the usability of a DOM node to the full degree, the Distribution Manager would have to send event handlers that has been attached with the DOM *addEventListener()* method. As a consequence of doing that, the method would become loophole for sending malicious code to other users. Attackers can exploit this vulnerability either by getting direct access to the distribution proxy or by intercepting traffic in the underlying network. Access to the distribution proxy can be guarded with an authorization mechanism and traffic interception can be avoided by using secure networking protocols. For Web applications neither of these provides adequate security however – all popular Web browsers have built-in Web development tools, with command line prompts able to execute arbitrary JavaScript code. With these, any user with access to the distribution proxy would have the ability to inject code into other users' computers with the *addEventListener()* function. A typical way of safely executing remote code is to letting it execute with limited access to the local machine (sandboxing) – for the Web there exists a number of projects that deals with sandboxing, such as Google Caja [17] and ADsafe [18], that lets guest code only use a safe subset of JavaScript. These does not answer the needs of a Distribution Manager with support for event handler distribution however, as they cannot validate code arriving from other devices dynamically on the fly. Moreover, sandboxing will be too restrictive as event handlers must be allowed to be quite expressive in order to function correctly. Another course of action would be to filter on keywords in the code, but this can be bypassed quite easily by code obfuscation.

Even though event handler distribution cannot be implemented in a secure way, there are reasons to argue that such distribution is not very useful anyways. A solution capable of automatically distribute event listeners, would have to opportunistically transfer all attached event handlers it comes across. The problem is that some events handlers are only meaningful in their original contexts, and thus inappropriate to distribute. As an example, they may reference parts of the UI that only exists on the device they were first registered. Certain styles of coding may also become an obstacle – more often than not, functions are not self-contained i.e. they reference variables outside the scope the function, and access to these will disappear when distributed. For these reasons, a DUI framework must have functionality for the programmer to decide which functionality that should be transferred, and which that should stay put, in order to define meaningful UI divisions. Among earlier work, Fröberg et al [5] probably emphasizes this the most, with their model for component callback coupling, where decoupled callbacks has a direct implementation in my library. Similarly, although not strictly related to event handlers, the frameworks by both Melchior et al [3] and Grolaux [4], a widget has different parts that either always stay on the same device or is transferred to other devices: the Widget Proxy that is stationary, and the Widget Renderer moves between devices.

User interaction concurrency

User interface components mirrored on multiple devices introduces subtleties that is not apparent in traditional single-user interfaces. Widgets are seldom designed to be used with multiple interaction modalities e.g. a slider widget is designed to be used with a single cursor, and not by two simultaneously. Ensuring that this is consistent is clearly an area where DUI toolkits can be of great help.

Google Docs [19] has such features e.g. when editing spreadsheets: a cell that is edited by one user is locked for others. Similarly, Luyten and Coninx [20] has given this problem attention and suggests a lock-based solution. Locks are far from trivial to implement however, as they have the potential of introducing problems such as deadlocks and data inconsistencies.

State synchronization

In the Distribution Manager library, shared UI state is implemented with DOM tree synchronization. This is not the only option – in the co-browsing solution by Lowet and Goergen [9] synchronization is based on JavaScript engine input i.e. by synchronizing UI events. The approach in a nutshell is to make sure that the exact order of events occurs in all browsers part of the co-browsing session. Compared to my solution, theirs is a bit different in that a Web page as a whole is synchronized rather than subsets of it. JavaScript engine input synchronization could possibly be integrated into the Distribution Manager library for mirrored nodes, but in the current implementation it is not. A deeper investigation on how this could be incorporated into my framework could be interesting – many of the problems encountered during the development is related to problems with inconsistencies in JavaScript and HTML state, and synchronization solely based on JavaScript state removes at least some of these problems.

Another way to synchronize UI state, is by doing it based on state variables provided by the programmer, such as in Panelrama [7] – state in Panelrama is handled very similarly to the way the developer is able to handle state with the *setState()* function in the Distribution Manager API.

Method

The goal of software frameworks/libraries is to make life easier for developers. Evaluation that this is the case for the Distribution Manager library has not been performed, which is a flaw of this work. The best way to investigate this would be a developer study that evaluates how intuitive the API of the library actually is. Even though this has not been performed, one can still try to draw conclusions in relation to the results from earlier work of similar systems. A developer study was performed on the Panelrama framework, with good results, and even though our approaches differs in many regards, they are similar in others. For the aspects that are similar, one can be cautiously optimistic that the Distribution Manager would perform well too.

Another way to evaluate the framework, is to measure how much programmers has to change their coding style compared to traditional UI development, and the increased development cost use of the library results in. The calculator case study showed that in that particular case, the development cost of using the library is very cheap. There is not enough data however, to conclude that this is true in the general case. It would therefore be desirable that more data was collected, so conclusions of the development cost can be evaluated more properly.

Regarding the code evaluation, it does not necessarily provide an accurate picture of the code quality. There are for example visibility issues with data members of the Distribution Object prototype, in that developers can access properties that should be internal to the library. This was the result of an early design decision, that later became to be hard change, even though the Maintainability Index indicates that cost change is low. Moreover, software modules with a small code base (such the library in this work), tends to get a better maintainability score, since it is correlated with number of lines of code.

Source criticism

There are currently only a few research groups that works on DUIs, which makes the selection of relevant research relatively easy compared to other fields. Earlier work on DUI toolkits specifically has been those of most interest however, rather than work on specific applications that makes use of distributed UIs for specific domains. Only a handful of DUI frameworks exists and many of them are presented in the theory section. For earlier work that revolves around Web technologies, the time of publication has been a factor when sorting out relevant papers. Web technologies has gone through an evolutionary explosion the last decade, which has made certain techniques used in some of the earlier work obsolete.

The work in a wider context

Hutchings and Pierce [21] concludes that users are interested in dividing their interfaces private, semi-private and public environments. Their results also suggest that users trust computing devices too much – a possible use case for DUIs in the future is for users to annex public devices, and this is likely to introduce security and privacy issues, which DUI developers must take into regard.

CONCLUSIONS

The research question in this thesis was how to implement distribution that preserves interactive functionality and how to synchronize shared UI state. The conclusions regarding these is that both can achieved to an at least acceptable degree without too much intervention required by the user of the Distribution Manager library.

In Web development interaction with the user is handled by attaching event listeners to DOM nodes, and my library has functionality similar to the native way of handling UI events – only minor changes in coding style is required to handle events in a distributed manner. The drawback of my solution is that it is sensitive to users leaving the DUI – to circumvent this limitation, users of the library can define initialization scripts that restores the event listener structure lost in the distribution.

Synchronization of shared UI state can in most cases be handled transparently by the library from the developers point of view. MutationObservers are used to track changes

in DOM trees and the library automatically updates corresponding DOM trees on other devices. In some cases DOM tree synchronization can lead to inconsistencies in HTML and JavaScript state, which is solved by letting the developer handle UI state manually with facilities provided by the library.

Future work

In the current implementation, it is assumed that resources such as images and stylesheets, are available at distribution target devices. Future work should implement something similar to the resource server in the Marve framework, so that shared resources are available to all clients.

The burden of adding distributed DOM nodes to a Web document in a functioning way is put on the developer. Other DUI toolkits are way more evolved in this regard, and further work should investigate how to incorporate adaptation mechanisms present in other toolkits, into the Distribution Manager API. This can for example be support for automatically choosing which devices that is appropriate to distribute to e.g. based on widget size/screen size ratio.

REFERENCES

[1] A. Demeure, J. Sottet, G. Calvary, and J. Coutaz, "The 4C reference model for distributed user interfaces models," *Auton. Syst. ...*, 2008.

[2] K. Bharat and L. Cardelli, "Migratory applications," *Proc. 8th Annu. ACM Symp.*, 1995.

[3] J. Melchior, D. Grolaux, J. Vanderdonckt, P. Van Roy, and P. Van Roy, "A toolkit for peer-to-peer distributed user interfaces: concepts, implementation, and applications," in *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, 2009, pp. 69–78.

[4] D. Grolaux, "Transparent Migration and Adaptation in a Graphical User Inter-face Toolkit," 2007.

[5] A. Fröberg, H. Eriksson, and E. Berglund, "Developing a DUI Based Operator Control Station," Springer London, 2011, pp. 41–49.

[6] R. Han, V. Perret, and M. Naghshineh, "WebSplitter: a unified XML framework for multi-device collaborative Web browsing web," *Proc. 2000 ACM Conf.*, 2000.

[7] J. Yang and D. Wigdor, "Panelrama: Enabling Easy Specification of Cross-Device Web Applications," *Proc. 32nd Annu. ACM Conf. Hum. Factors Comput. Syst. - CHI '14*, pp. 2783–2792, 2014.

[8] S. Firmenich, G. Rossi, M. Winckler, and P. Palanque, "An approach for supporting distributed user interface orchestration over the Web," *Int. J. Hum. Comput. Stud.*, vol. 72, no. 1, pp. 53–76, 2014.

[9] D. Lowet and D. Goergen, "Co-browsing dynamic web pages web," *Proc. 18th Int. Conf.*, 2009.

[10] J. Lo, E. Wohlstadter, and A. Mesbah, "Imagen: runtime migration of browser sessions for javascript web applications web," *22nd Int. Conf. World ...*, 2013.

[11] J. Kwon, J. Oh, I. Jeong, and S. Moon, "Framework separated migration for web applications web," *Embed. Syst. Real-time Multimed.*, 2015.

[12] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *Proceedings Conference on Software Maintenance 1992*, pp. 337–344.

[13] "HTML Standard." [Online]. Available: https://html.spec.whatwg.org/multipage/comms.html#network. [Accessed: 10-Apr-2017].

[14] "DOM Parsing and Serialization." [Online]. Available: https://w3c.github.io/DOM-Parsing/#introduction. [Accessed: 11-Apr-2017].

[15] "W3C DOM4." [Online]. Available: https://www.w3.org/TR/dom/#mutationobserver. [Accessed: 11-Apr-2017].

[16] "JavaScript source code visualization, static analysis, and complexity tool." [Online]. Available: https://github.com/es-analysis/plato. [Accessed: 11-May-2017].

[17] "Introduction | Caja | Google Developers." [Online]. Available: https://developers.google.com/caja/. [Accessed: 20-Apr-2017].

[18] "ADsafe." [Online]. Available: http://www.adsafe.org/. [Accessed: 12-Apr-2017].

[19] "Google Docs - create and edit documents online, for free." [Online]. Available: https://www.google.se/intl/en/docs/about/. [Accessed: 11-Apr-2017].

[20] K. Luyten and K. Coninx, "Distributed user interface elements to support smart interaction spaces," *Multimedia, Seventh IEEE Int. Symp.*, 2005.

[21] H. H. M. Hutchings and J. J. S. Pierce, "Understanding the whethers, hows, and whys of divisible interfaces," in *Proceedings of the working conference on Advanced visual interfaces*, 2006, pp. 274–277.