UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of computer science
Software engineering curriculum

Madis Nõmme

Implementing in-browser screen sharing library for robust, high-performance co-browsing

Master's thesis (30 EAP)

Supervisor: Satish Narayana Srirama, PhD

Tartu 2015

Implementing in-browser screen sharing library for robust, high-performance co-browsing

Abstract:

Co-browsing is the activity of two or more people viewing and interacting with the same web page simultaneously using web browsers running on different computers. It can be useful for many purposes: entertainment, information sharing, learning, supervision, surveillance, etc. In this work the author concentrates on the issues and challenges in implementing co-browsing for assisting users.

Sharing user's screen has historically required installation of additional software. Installing such software often requires elevating user privileges and can introduce security risks. Also separate implementations on different operating systems make it more expensive to develop. Many application developers have discovered browsers as their new *platform independent* platform. As more applications are moving into the browser, it has become a good candidate to implement screen sharing on. Screen sharing in a browser is called co-browsing.

This thesis describes a solution for screen-sharing inside browser without any plugins or additional software.

Keywords:

co-browsing, screen sharing, desktop sharing, real-time communication, peer to peer, WebRTC

Kõrge jõudluse ja veakindla ühisbrausimise teegi arendamine veebisirvikutes ekraani jagamiseks

Lühikokkuvõte:

Ühisbrausimine on tegevus, mille käigus kaks või enam inimest näevad sama veebilehte erinevate arvutite tagant samaaegselt ja suhtlevad teineteisega hiireklikkide, kerimise ja muude juhtseadmete toimingute kaudu. Selline tegevus võib olla kasulik erinevatel põhjustel, nagu meelelahutus, teabe jagamine, õppetöö, juhendamine, järelvalve jne. Käesolevas töös keskendub autor küsimustele ja väljakutsetele, mis on seotud ühisbrausimisega kasutajate abistamise eesmärgil.

Ekraanijagamine nõuab tavaliselt eriotstarbelise tarkvara paigaldamist. Sellise tarkvara paigaldamine on tihti seotud kõrgentatud kasutajaõigustega, mis omakorda võib põhjustada turvariske. Iga operatsioonisüsteemi jaoks eraldi programmikoodi kirjutamine muudab sedasorti tarkvara arendamise kulukaks. Seetõttu on paljud rakenduste arendajad oma tooted veebisirvikutesse ümber kolinud. Kuna järjest rohkem rakendusi kirjutatakse veebisirvikus käitamiseks, siis on viimase veetlus ühisbrausimise platvormina suurenenud. Ekraani jagamist veebisirvikus nimetatakse ühisbrausimiseks.

See töö kirjeldab tarkvaralahendust veebisirvikutes ekraani jagamiseks ilma ühegi lisamooduli või täiendava tarkvarata.

Võtmesõnad:

ühisbrausimine, ekraani jagamine, töölaua jagamine, suhtlus reaalajas, otsesuhtlus, WebRTC, veebisirvik

Contents

List of Tables

List of Figures

1 Introduction

We use computers in plethora of our daily activities. Some of them are as simple as checking a new incoming email notification. Others are more complex - like designing a model for 3D printer or making the decision to buy certain stock or insurance policy. Browsers that were once mainly used for simple applications targeted for information and media consumption are being target platform for complex and useful applications for which the user needs to be trained or have moderate amount of experience in order to handle the system and make correct decisions.

Traditionally helping confused users has been done in written form or via phone support. It requires certain levels of frustration and motivation to make user find and read the manual or grab a phone to call help desk. Often people just give up and leave the website.

To keep people engaged and solve their problems on the spot, the support service agent needs to understand the circumstances of the user. When providing support through a website, the traditional approach has been to describe the issue using text-chat. Most people are not very proficient in entering text using keyboard and the interaction can quickly become cumbersome. Better solution would be to make the communication interactive so that the user can demonstrate the issue to the helping person right on the page.

Often the interactive assistance tools are used by system administrators when configuring remote computer or providing remote help to a user. Microsoft Windows has included support for Remote Desktop protocol since Windows XP[1], OS X has done the same with their software called Screen Share since OS X 10.5 (Leopard)[2]. These solutions work in a setting where you trust your assister and both sides have necessary software installed. It is not suited for situations where there are:

1. many customers coming and going. It can be difficult for the assisting operator to decide which user to contact. Also many of the users might be on the page for the first and last time so no information is known about them. The classic screen sharing tools do not offer any metrics for measuring user confusion.

2. customers are using different platforms (PC, Mac, Mobile). To assist them, they

would have to have the screen sharing software installed allow themselves to be assisted. This does not work in untrusted setting - random web page and introduces security issues.

Requirements for the ideal tool would be:

1. no installation on the client side
2. support for passive observation for the operator without any interaction required from visitor
3. works in browsers so platform independence is achieved

1.1 Outline

The State of the art chapter gives an overview of the existing technologies and approaches in screen sharing and co-browsing realms. Operating system level screen sharing applications use pixel data to transport visual screen information to the other participant. In-browser solutions can use the pixel approach (as in the *canvas* and *webrtc* approaches) but more flexible results can be achieved through what is called structured content rendering. It works by taking DOM representation from one browser and reconstructing it in another.

Goals and issues to solve chapter describes which co-browsing issues a library must solve and unique features it has to implement in order for it to stand out among other analogous software solutions. *Dynamic features* allow to package groups of functionality into modes and switch between these modes during a co-browsing session. *Content filtering* helps to omit or mask parts of the page that co-browsing is happening on. Examples of omitted or masked page parts are personal information like credit card numbers or a DIV element containing picture of the user. *Navigation and page reloads* need to be dealt with to make co-browsing look smooth and uninterrupted for the receiving side. If user is navigating, the WebSocket connection gets disconnected between page reloads, so the library needs to know that the user is not actually leaving page. *User presence and disconnects* is related to the navigation disconnects but also has to deal with notifying the end of co-browsing session in case the user is idle for too long. *Visual outlook* section describes why it is necessary to deal with different and often obscure ways that pages use CSS and how to make injected CSS to not conflict with the page. *Latency and*

responsiveness describes why it is important to optimize for latency and technologies to use for it (WebSockets, WebRTC data channels). *Security issues* are an important concern of an injected 3rd party JavaScript library that dynamically transports web page content between users. Different MITM and cross site scripting issues could arise.

Implementing Cobra chapter describes parts of the implemented co-browsing library. Cobra implementation follows plugin architecture. There are two sides: *Source*, which collects and forwards co-browsing data and *Screen* which takes info sent by *Source* and transforms it into displayable form. Basic co-browsing is supported by *Drivers*. *Screen* side runs *DriverMaster* and source side runs *DriverSlave* Two driver implementations - *UrlDriver* and *MutationSummaryDriver* are described. Additional functionality like support for keyboard, pointers, focus and scrolling is implemented by *Features*. Different features can be grouped together into *Modes*. The *Co-browsing process* section describes how the library setup process works. To allow other communication forms like chat and audio/video calls persist during navigation, the co-browsable page must be iframed. Iframing, switching modes and teardown are also described in this chapter.

Evaluating Cobra chapter describes testing approach and some performance characteristics of the devised co-browsing library.

2 State of the art

In general, screen sharing software can be categorized in based on the platform they run on: operating system, inside application, browser or based on how the screen representation gets built for the other participant: pixel info, structured content rendering.

2.1 Screen sharing based on platform

Platform here means the collection of API-s and the provider of these, that is used to achieve screen sharing.

2.1.1 Operating system

Operating system screen sharing requires a special program to be installed. Sometimes the program can be bundled with the operating system[1]. These applications provide greatest level of control and pixel-perfect replication of user's screen. They require extra effort and knowledge from the user to install them. Also they give the joining party permissions to access the whole computer with the same rights as the user, which can cause trust and security issues. Operating system based screen sharing tools are best for providing temporary assistance and aid in system administration.

2.1.2 Application specific

Great example of application specific screen sharing are the Google Drive suite applications - Docs, Sheets, Slides, Forms, Drawings. Each of these applications provides specific functionality and adds co-browsing where users can collaborate within the bounds of the specific application's functionality.

2.1.3 Browser based

Browser based screen sharing means that the code that implements the logic for getting one user's screen visible to another runs in browser and uses browser provided API-s

[1] Apple Screen Sharing

to achieve its goal. Browser based screen sharing gets number of benefits from its host environment:

- runs on the same platforms that the browsers run
- has access to standardised networking API-s
- gets browser provided sandboxing, mitigating many security issues
- has access to structured contents used for rendering the application
- once implemented, works on most of the *web applications*

2.2 Sharing screen based on image reconstruction

2.2.1 Pixel info based screen sharing

Image based screen sharing implementations are possible both on operating system and inside browsers. They work by capturing image representation from a source e.g. a frame buffer or HTML canvas element. In the case of widespread standard of operating system screen sharing protocol VNC, reproducing the screen image works through *"put a rectangle of pixel data at a given x,y position"*[3].

Where in Operating System (OS) level a framebuffer can be accessed, in browsers different mechanisms exist to achieve the same goal. Because the representation of user's browser comes in as a flat image with these methods, no direct interaction with the page is possible on the *screen* side.

To provide interactive control to the *screen* user, events from input devices have to captured and triggered on the *source* user's computer. The resulting visual changes then arrive to the *screen* user's browser over the used mechanism (canvas, webrtc).

2.2.2 Using the *canvas* element

The canvas element provides scripts with a resolution-dependent bitmap canvas, which can be used for rendering graphs, game graphics, art, or other visual images on the fly[4].

The *canvas approach* relies on two techniques:

1. It is possible to draw web page contents on `<canvas>` element.

2. It is possible to convert the contents of HTML5 `<canvas>` element to image data.

Example code for capturing & sending image data from `<canvas>` element:

```
function blobFromDataUrl(dataUrl) {} // http://git.io/vUyJg
function makeWebsocketConnection() {} // Stub

// Sender:
var canvasEl = document.getElementById('some-canvas-element')
wsocket = makeWebsocketConnection()
wsocket.binaryType = 'blob';
wsocket.send(blobFromDataUrl(canvasEl.toDataURL()));

// Receiver:
function receiveAndDisplayBlob(blob) {
  imageEl = document.getElementById('image-target');
  imageEl.src = URL.createObjectUrl(blob)
}
wsocket.addEventListener('message', receiveAndDisplayBlob)
```

More sophisticated approach (like the *Html2Canvas* by Niklas von Hertzen) have to use approach where they render the current page as a canvas image, by reading the DOM and the different styles applied to the elements[5].

The main drawbacks of the *canvas* approach are

1. Not all CSS caused visual effects are captured
2. It takes time & resources to capture the page and construct the image Blob
3. When co-browsing parties are using different resolutions, the image can become stretched
4. Result is static image. So any animations are omitted

Benefits of the *canvas* technique are that that the generated image looks visually very similar with on matching resolutions. The image can also be saved on the server side.

Since the image layer is flat, there is no way to alter or interact with its contents. Any user input must be sent to the source and visual changes caused by it transferred back through the *canvas* mechanism.

2.2.3 Using WebRTC

WebRTC is a free, open project that provides browsers and mobile applications with Real-Time Communications (RTC) capabilities via simple APIs. The WebRTC components have been optimized to best serve this purpose[6].

WebRTC and related API-s are implemented on Google Chrome and Mozilla Firefox browsers and their derivatives.

The main objective of WebRTC is enabling audio and video calling between browsers. In addition to camera being the media source, users can request *desktop* as video source.

Requesting *desktop* as video source:

```
constraints = {audio: false, video: {mandatory: {
    chromeMediaSource: 'desktop',
    chromeMediaSourceId: thestreamid,
    maxWidth: window.screen.width,
    maxHeight: window.screen.height,
    maxFrameRate: 3,
}}};
getUserMedia(constraints, callback);
```

It allows capturing user's desktop, individual windows or browser tabs as video stream source. The video stream is compressed for transfer using VP8[2] or H.264[3] encoding[7].

[2]VP8 is a video compression format owned by Google and created by On2 Technologies as a successor to VP7

[3]H.264 or MPEG-4 Part 10, Advanced Video Coding (MPEG-4 AVC) is a video compression format that is currently one of the most commonly used formats for the recording, compression, and distribution of video content

2.3 Structured content rendering

Browsers hold the web page in a DOM[4] representation in memory. The *structured content rendering* method relies on sending information from *source* to *screen* so that the later one could reconstruct the DOM and thus render the original web page.

To render a web page the browser needs to combine CSSOM[5] and DOM trees into a render tree, which is then used to compute the layout of each visible element and serves as an input to the paint process which renders the pixels to screen[8].

The most compact form to communicate existing DOM from one user to another is by sending over the URL that current page is fetched. If the server doesn't require cookies or other information that is not present in the URL for returning the page, the receiving user will get the same HTML content as the other user. Receiving user's browser will render it to look very similar.

Another way of getting the DOM from one user's browser to another is to serialize the DOM and send it over the network. This is more complex than just calling `JSON.serialize(document)` as the resulting data structure has to be JSON serializable (can not contain cycles) and it has to include all the necessary data for reconstructing the DOM on the receiving browser.

Pseudocode for serializing DOM:

```
serializeNode = (node) ->
  data = extractGeneralNodeData(node)
  addNodeTypeSpecificData(node, data.nodeType)
  if node.nodeType == Node.ELEMENT_NODE
    addSerializedChildrenData(node, data)
```

Pseudocode for de-serializing DOM:

```
deserializeNode = (nodeData, parent, root) ->
  doc = root.ownerDocument || root
```

[4]Document Object Model
[5]CSS Object Model

```
# Node types e.g. Node.DOCUMENT_TYPE_NODE or
# Node.ELEMENT_NODE etc.
node = createSpecificNodeType(nodeData.nodeType)
parent.appendChild(node) if (parent)
if nodeData.childNodes?
  deserializeNode(child, node) for child in nodeData.childNodes
return node
```

Serializing and sending the whole DOM every time part of it changes is not practical. There exist API-s for observing when DOM changes and getting parts of the changed DOM.

Mutation Events is an older API and is deprecated in newer browsers. It is slow and synchronous. User gets updated every time a change happens. It performs worse than the newer API **Mutation Observers**.

Mutation Observers is the new asynchronous API for observing DOM changes. It can be attached to a DOM node and will provide the developer change type and changed attributes including same information for child nodes.

2.4 Other useful technologies

Regardless of how the data describing user's screen visual state and input devices is collected, it needs to be sent to the other peer. Although using HTTP requests is possible, the HTTP protocol adds quite a bit overhead introduces latency issues. Mainly because one side would have to post to a central server and another would request the data. More efficient ways are WebSockets.

2.4.1 WebSockets

All modern browsers and Internet Explorer since version 10 support WebSockets. WebSocket is a protocol providing full-duplex communication channels over a single TCP connection. The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011, and the WebSocket API in Web IDL is being standardized by the W3C[9]. WebSockets allow real time messaging with very little overhead.

2.4.2 WebRTC data-channels

WebRTC is a relatively new standard for exchanging real time media in browsers. It's value for co-browsing in the context of mechanisms is what are called *data channels*.

WebRTC sends media streams from browser to browser in peer-to-peer manner. The *data channels* part of the api enables to send the co-browsing data without using central server or hub. Thus getting rid of half of the latency.

Having both sides to exchange co-browsing data directly has the following benefits:

1. Reducing load on the server infrastructure.
2. Providing better responsiveness due to smaller latency as the packets travel directly between two browsers without going through a relay WebSocket server.

2.5 Existing commercial and free implementations

There are some existing companies and projects that offer co-browsing as part of their functionality. 7 categories were taken as a basis of comparison.

1. *Method* is the implementation type of co-browsing.
    - Mutation - a MutationSummary based implementation
    - Image - an implementation based on encoding the observed page into image and
    - Url - URL of the co-browsed page is sent to the other participant
2. *Mouse* shows whether user's mouse pointer is shown to none, one side or both.
3. *Keyboard* describes whether keyboard events were sent and played back to the other user.
4. *Forms* is about syncing forms after non-user induced DOM change. This includes re-filling forms after iframing the page.
5. *1 line itegration* means whether the simplest way of integrating was supported - inserting 1 script tag with src attribute containing the integration script.
6. *Mobiles* support means working without significant visual or functional errors on mobile devices. Safari browser on iPhone4 was used for testing.

Table 1: Comparison of existing implementations and libraries

Name	Method	Mouse	Keyboard	Forms	1 line integration	iframing	Mobiles
Cobra	All	2-way	2-way	Yes	Yes	Yes	Yes
Surfly	Mutations	1-way	1-way	No	No	No	No
Kandy	Image	No	No	No	No	No	No
Firefly	Mutations	2-way	No	No	No	No	No
TogetherJS	Url	2-way	2-way	No	Yes	No	Yes

1. Cobra - supports all the compared features
2. Surfly (https://www.surfly.com/) is a product supporting co-browsing and video. It passive co-browsing only meaning that one user will be sent to a page where he can passively observe what the other user does. Scrolling did not work on mobile devices. They build a mirror page on their own domain. This means that the product is not working directly on the potential clients' site.
3. Kandy (https://www.kandy.io/) implements image based screen sharing. No iframing is done. Kandy works only on one page and breaks after navigation. No keyboard or mouse support was present.
4. Firefly (http://usefirefly.com/) used MutationObservers approach. Sent mouse mouse but not keyboard. Establishing co-browsing session was cumbersome - four manual actions were needed (including the user having to send support ID to the support person) before the session could begin. Only sent mouse pointer one way. It did not support iframing which would make functionality that has to persist between navigation hard to support.
5. TogetherJS (https://togetherjs.com/) supported only URL based co-browsing. Only

product besides Cobra that supported 1 line integration. It relies on the co-browsed application syncing its state between multiple simultaneous users. Provides only URL syncing and mouse pointer forwarding.

5 Evaluating Cobra

5.1 Performance

The apparent responsiveness of Cobra's current implementation depends mainly on two factors: network bandwidth and network latency.

To counter network latency, the DynamicChannel switches to WebRTC data channels whenever possible. This avoids the messaging traffic going through a central server and instead flows directly between browsers.

An experiment was also conducted to measure CPU load added by Cobra. The experiment was set up as follows:

1. DOM tree of depth 10 with 2046 nodes was generated
2. A repeating function that swaps 10 DOM nodes every second
3. Measure CPU load without co-browsing
4. Measure CPU load with co-browsing

Table 3: CPU load with and without co-browsing

	Screen	Source
Idle	4.6%	5.1%
With co-browsing	9.9%	7.0%

The test was conducted on a Mid 2013 Macbook Air with 1.7 GHz i7 CPU, base CPU load of 12%. The DOM changes frequency in the test is greater than real user activity. Real human caused DOM activity happens in spikes (i.e. it isn't constant load) when navigating to a new page or opening a subsection from an application. Author considers the increase in CPU load, seen from the test results in Table 3 too small to affect usability in any negative way.

5.2 Browser compatibility

Browser compatibility is definitely a concern for Cobra, as it has to work on almost any browser. Because of the universal support for the API used by UrlDriver, Cobra is usable on all recent browsers. As of April 2015, 97.7% of users are using a browser that supports Cobra[9]. Comparison of support for various features used by Cobra can be seen from Table 4.

Table 4: Browser compatibility of technologies supporting co-browsing

Browser	Mutation Events	WebRTC	UrlDriver	Mutation Summary
Google Chrome	Yes	Yes	Yes	Yes
Mozilla FireFox	Yes	Yes	Yes	Yes
Internet Explorer 9	Yes	No	Yes	No
Internet Explorer 10	Yes	No	Yes	No
Internet Explorer 11	Yes	No	Yes	Yes

5.3 Testing

One of the goals of building Cobra was it to be modular and testable. The tests are written using Mocha[10] testing framework.

Many parts of Cobra have two sides: source and screen. Because of this it is often useful and more efficient to not strictly unit test but to write integration tests instead.

5.3.1 Testing drivers

Current implementation of Cobra has two drivers: UrlDriver and MutationSummaryDriver.

UrlDriver's only responsibility is to send the *window.location.href* to the other side. In the test source iframe is navigated to an url, after a short delay assertion against display side's location will be made.

[9] http://www.w3schools.com/browsers/browsers_stats.asp
[10] http://mochajs.org/

MutationSummaryDriver has to send initial dom and after that dom changes. The generic way of testing it is working quite well: add an element to the source iframe's DOM and after a short delay observe that change in display side's DOM.

5.3.2 Testing features

All Features contain two components: source and display. Because of this property it is more useful to test the interaction of both parts instead of unit testing each separately. The basic testing flow is as follows:

1. Create 2 iframes, one with ad-hoc DOM
2. Create the display part of the object. Attach it to iframe 1
3. Create the source part of the object. Attach it to iframe 2
4. Connect source & display parts over ad-hoc channel
5. Simulate change in DOM on the source side
6. Assert feature specific result on the display side

6 Conclusions

Co-browsing is screen sharing in browsers. Browsers can be used as a platform for developing universal co-browsing functionality that works on all web pages. Developing the library in browser has additional benefit of being available on all the platforms that the browser is, getting standardized networking API support and having access to the structured contents of the web page (DOM).

The thesis investigated different requirements of a good co-browsing library and compared number of currently available solutions. All existing solutions are missing features or are unstable in some common scenarios. A new library, called Cobra was proposed and developed.

Cobra implements co-browsing through Drivers and Features. Drivers get the visual representation of the page to another user. Features add interactivity like mouse cursor and keyboard support. This separation allows easier way of reasoning, testing and configuring the functionality.

Cobra is part of the SaleMove platform. Together with *Multicom*[11] they make up the interactive communications part of the system. As of May 2015, there have been ~33127 co-browsing sessions with average duration of 7.5 minutes.

[11] Multicom is unified in-browser video, audio and phone calling library of SaleMove, leveraging the latest WebRTC technology, developed also by the author.

7 Future improvements and ideas

The chosen approach for developing Cobra has proven to be very successful. The implementation is stable and has been in production use for almost one year (as of May 2015). It has been easy to understand and explain to new developers and issues can be reasoned about and localized quickly because of the modular approach.

Some ideas for making Cobra even more useful and more featured would be:

- Implementing recording & playback of co-browsing session. Users might be interested in reviewing their interaction. This feature can be useful also for debugging reasons.
- Implementing other drivers besides the current UrlDriver and MutationSummaryDriver. The CanvasDriver and WebRtcDriver could provide some interesting capabilities. CanvasDriver would allow taking screenshots very easily. WebRtcDriver's current holdback is that it requires the installation of chrome extension. When implemented however, it can provide pixel perfect representation of users browser tab.
- Group co-browsing would be interesting addition. It could allow teacher-class kind of setups where one person broadcasts his screen and multiple people join to see.

References

[1] Wikipedia, *Remote desktop protocol - wikipedia, the free encyclopedia.* [Internet]. http://en.wikipedia.org/wiki/Remote_Desktop_Protocol.

[2] Wikipedia, *Screen sharing - wikipedia, the free encyclopedia.* [Internet]. http://en.wikipedia.org/wiki/Screen_Sharing.

[3] T. Richardson et al., *Virtual network computing*, *Internet Computing, IEEE*, kd 2, nr 1, lk 33–38, Jan 1988.

[4]*HTML living standard*, May-2015. [Internet]. https://html.spec.whatwg.org/multipage/scripting.html#the-canvas-element.

[5] N. von Hertzen, *HTML to canvas*, May-2015. [Internet]. https://github.com/niklasvh/html2canvas/blob/master/readme.md.

[6] google chrome team, *WebRTC.* [Internet]. WebRTC is a free, open project that provides browsers and mobile applications with Real-Time Communications (RTC) capabilities via simple APIs. The WebRTC components have been optimized to best serve this purpose.

[7] Infoworld, *Browsers must support h.264 and vP8 as part of real-time communications effort.* [Internet]. http://www.infoworld.com/article/2847248/web-browsers/webrtc-compromise-on-video-codec-standards.html.

[8] I. Grigorik, *Render tree construction, layout and paint.* [Internet]. The CSSOM and DOM trees are combined into a render tree, which is then used to compute the layout of each visible element and serves as an input to the paint process which renders the pixels to screen. Optimizing each of these steps is critical to achieve optimal rendering performance.

[9] W. contributors, *WebSocket*, *Wikipedia*, 2015. [Internet]. http://en.wikipedia.org/wiki/WebSocket.

[10] M. developer network, *Window.postMessage().* [Internet]. https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage.

[11] Wikipedia, *JQuery - wikipedia, the free encyclopedia.* [Internet]. http://en.wikipedia.

org/wiki/JQuery.

[12]*Reactive-extensions/RxJS*, May-2015. [Internet]. https://github.com/Reactive-Extensions/RxJS.

[13] Wikipedia, *Underscore.js - wikipedia, the free encyclopedia*, May-2015. [Internet]. http://en.wikipedia.org/wiki/Underscore.js.

[14]*Socket.IO - wikipedia, the free encyclopedia*. [Internet]. http://en.wikipedia.org/wiki/Socket.IO.

Extras

I Terms used

Co-browsing

The activity of sharing screen in a browser

Source

the user of the co-browsing process whose page is being co-browsed. This side produces the co-browsing updates. Also the object that governs DriverSlave and FeatureManager in that browser on the source side.

Screen

the side that displays the co-browsing updates. Also the object that governs DriverMaster and FeatureManager in that browser on the screen side.

II Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Madis Nõmme (date of birth: 2. mai 1984)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:

   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and

   1.2. make available to the public via the university's web environment, including via the DSpace digital archives, as of May 1st, 2017 until expiry of the term of validity of the copyright, **Implementing in-browser screen sharing library for robust, high-performance co-browsing** supervised by **Satish Narayana Srirama, PhD**

2. I am aware of the fact that the author retains these rights.

3. This is to certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, May 21, 2015