Webstrates: Shareable Dynamic Media

Clemens N. Klokmose[1], **James R. Eagan**[2,3],
Siemen Baader[1], **Wendy Mackay**[4,5,3] **& Michel Beaudouin-Lafon**[5,3,4]

[1]Aarhus University, [2]Télécom ParisTech, [3]CNRS, [4]INRIA, [5]Université Paris-Sud
clemens@cs.au.dk, james.eagan@telecom-paristech.fr, sb@cs.au.dk, {mackay, mbl}@lri.fr

Figure 1: Sample uses of *Webstrates*: (a) Collaborative document authoring with different editors personalized at run-time; (b) Multiple devices used to sketch a figure (tablet 1), see it in a print preview (tablet 2), and adjust it in a graphics editor (laptop). (c) Distributed talk controlled remotely by a speaker with a separate interface for audience participation.

ABSTRACT

We revisit Alan Kay's early vision of dynamic media that blurs the distinction between documents and applications. We introduce shareable dynamic media that are *malleable* by users, who may appropriate them in idiosyncratic ways; *shareable* among users, who collaborate on multiple aspects of the media; and *distributable* across diverse devices and platforms. We present *Webstrates*, an environment for exploring shareable dynamic media. *Webstrates* augment web technology with real-time sharing. They turn web pages into substrates, i.e. software entities that act as applications or documents depending upon use. We illustrate *Webstrates* with two implemented case studies: users collaboratively author an article with functionally and visually different editors that they can personalize and extend at run-time; and they orchestrate its presentation and audience participation with multiple devices. We demonstrate the simplicity and generative power of *Webstrates* with three additional prototypes and evaluate it from a systems perspective.

Author Keywords

Real-time Collaborative Documents; Dynamic Media; Web.

ACM Classification Keywords

H.5.2 User Interfaces.

INTRODUCTION

In the seventies, Alan Kay introduced the concept of *Personal Dynamic Media* that let a user "mold and channel its power to his own needs" [15]. He envisioned children with linked Dynabooks tinkering with a Spacewar game to make it more challenging by adding a more sophisticated form of gravity [14]. Two decades later, Mark Weiser envisioned a future of ubiquitous computing [30], where heterogenous devices of varying sizes and capabilities interact easily with each other and technology disappears into the background. He imagined how colleagues would share a virtual office and collaborate on a document, seamlessly moving between a wall-sized display and various 'tabs' and 'pads.'

Today, the hardware aspects of Kay's and Weiser's visions have been largely realized in smartphones, tablets, laptops, and large displays. Unfortunately, software lags far behind. Most software is brittle or hard to change: end-users are limited to sets of pre-determined lists of preferences, and even trained developers are at the mercy of application developers to provide hooks for adding or changing functionality.

Although documents are easy to exchange, they are difficult to *share*: Cloud-based file synchronization is prone to conflicts; Real-time collaborative editors are limited to a few application domains and require users to migrate their documents from their familiar tools to a new environment; Sharing applications is usually limited to bitmap-based screen sharing. Finally users must learn different environments as they shift from applications on laptops to apps on phones and tablets. The web partially mitigates the latter problem, but at the cost of limited functionality, interaction and extensibility.

Our vision of *shareable dynamic media* embodies three key properties:

- *Malleability*: users can appropriate their tools and documents in personal and idiosyncratic ways;

- *Shareability*: users can collaborate seamlessly on multiple types of data within a document, using their own personalized views and tools; and
- *Distributability*: tools and documents can move easily across different devices and platforms.

In this paper we present the concept of shareable dynamic media, an extension of Kay's dynamic media that supports "true" sharing and that challenges the traditional model of applications and documents, and we introduce Webstrates, which uses conceptually simple but powerful changes to the web infrastructure to implement these ideas and demonstrate their potential. We begin with a brief description of the concepts of shareable dynamic media and substrates followed by an introduction to Webstrates. We compare Webstrates to related work and demonstrate both its power and simplicity through two case studies and three examples, including the collaborative authorship of this paper. We describe the implementation and evaluation of Webstrates and discuss both the limitations and potential of this approach.

THE WEB AS SHAREABLE DYNAMIC MEDIA

We define shareable dynamic media as collections of *information substrates* (or substrates for short). Substrates are software artifacts that embody content, computation and interaction, effectively blurring the distinction between documents and applications. Substrates can evolve over time and shift roles, acting as what are traditionally considered documents in one context and applications in another, or a mix of the two. Substrates may be composed in various ways, e.g., one substrate can give meaning and structure to another. For example, a bar-chart substrate can define how to visualize a statistical data substrate.

To create a malleable substrate that blurs the distinction between development and use, as in Kay's vision of children tinkering with their game together, we also need substrates to be truly shareable and to support changes to their content, computation and interaction elements at run-time. Dynamic shareable media therefore require support for:

- representation of various content types, including text, images, diagrams, video, and sound;
- real-time sharing of content across a variety of devices;
- persistent storage of content as it changes;
- embedded computation within the substrate itself to create and manipulate content; and
- on-the-fly recombination of content and computation.

In order to experiment with the concept of shareable substrates, we needed to create a prototype that is both sufficiently rich yet simple to develop, and easy to deploy on various devices. The web offers an appealing platform. Web pages are globally addressable with URLs, and can represent rich content that mixes text, images, etc. in a platform-independent way. Content is represented both in an easy-to-store text format (HTML) and as objects in a standardized Document Object Model (DOM). The DOM can be manipulated at run-time with JavaScript, and changes are immediately reflected on the display.
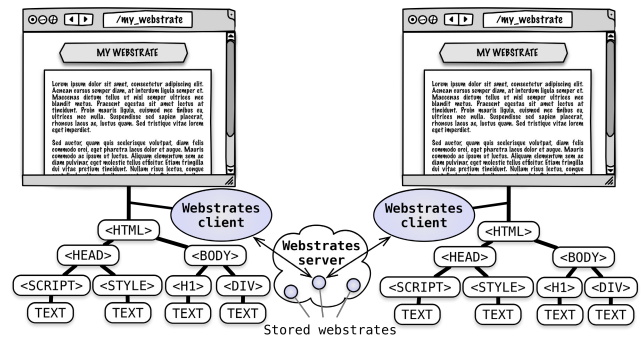

Figure 2: Opening the same webstrate in different browsers. Changes to the DOM are synchronized among clients and made persistent on the *Webstrates* server.

Finally, the performance of web browsers and JavaScript interpreters is constantly improving, with a large number of web programming tools and libraries.

Unfortunately, the web is not designed to support sharing: client-side changes made to a web page do not persist, nor are they synchronized with other clients of the same page. The web also does not natively support content manipulation: content must be edited with separate applications, or by embedding ad hoc code within pages and on the server.
We must thus augment the web with fundamental support for persistence and synchronization of client-side changes to web pages in order to support malleability, shareability, and distributability.

We introduce the *Webstrates* system (web + substrates), a prototype of shareable dynamic media that consists of a custom web server that serves pages, called webstrates, to regular web browsers. Each webstrate is a shared collaborative object: changes to the webstrate's DOM, as well as changes to its embedded JavaScript code and CSS styles, are transparently made persistent on the server and synchronized with all clients sharing that webstrate, using Operational Transformations [8] (Fig. 2). By sharing embedded code, behavior typically associated with application software can also be (collaboratively) manipulated, opening the way to novel possibilities for combining content, computation and interaction. Webstrates can be composed by embedding one webstrate within another, a process called transclusion [22], that lets users truly share, rather than copy, content.

We illustrate how *Webstrates* works with two implemented case studies. Case study 1 illustrates collaborative paper authoring by Alice in Europe and Bob in the United States. Each use *personalized editors* on the same document. Using *instrumental interaction* [2], Alice has created a citation tool for her editor to add references. She performs *remote user interface extension at run-time*, opening Bob's editor and adding the citation tool to his toolbar. This is a *malleable interface*: Alice modifies Bob's editor to hide his toolbar and make it appear when he hovers over it. Bob performs *live tranclusion of figures*, sketching a rough figure on a tablet which updates in real time in the paper editor on his laptop as Alice corrects it with her vector-graphics editor.

Case study 2 illustrates Alice and Bob preparing and giving a talk. Alice creates a slideshow in an editor that contains many of the same tools as her paper editor, including her citation tool. During their *collaborative slideshow design*, Alice shares the slideshow with Bob, and navigates through it remotely as they discuss it via video chat. Bob adds notes and she updates the slides immediately. At the conference, Alice gives a *distributed slideshow presentation*. Audience members can see the slides on their personal devices and pose questions. The session chair selects three questions that are displayed on Alice's final slide.

The next sections present related work and the fundamental concepts of *Webstrates*. We then return to these two case studies and describe how each feature is implemented.

RELATED WORK

Prototypes of the Dynabook were written in Smalltalk [10], whose late binding enables the reprogramming of applications at run time and blurs the distinction between development and use. The current web run-time environment, and thus *Webstrates*, is also malleable, although scripts are not re-interpreted by default and changing them requires reloading the page. Also, the web is a universal distributed environment, a de facto standard that runs on all platforms. Futhermore, the advent of web applications is pushing web technology towards greater run-time flexibility, making it a viable alternative to environments such as Smalltalk.

Boxer [3] was an environment for children to learn to program by "controlling a reconstructible medium, much like written language, but with dramatically extended interactive capabilities." Text, graphics and code are all embedded in nested boxes (based on LISP) that can be displayed, modified and shared by both users and programmers, making it easy to create dynamic, interactive documents. Boxer's composition model is similar to the notion of transclusion in *Webstrates*.

Hypercard [11] let users create, manipulate and exchange stacks of interactive multimedia cards. The integrated authoring environment allowed both authors and end-users to add functionality, such as associating functions to buttons or adding interactive animations, effectively blurring the line between documents and applications. *Webstrates* makes it possible to create editors similar to Hypercard, with two important additional capabilities: native support for collaboration and tailorability of the authoring environment based on individual preferences.

More recently, the Lively Kernel [29] offers an integrated web browser development environment designed for creating desktop-style applications. It supports run-time software malleability, but is limited to asynchronous wiki-style collaboration [19]. Although web-based, it abstracts away from the DOM, an approach fundamentally different from *Webstrates*.

An important limitation of the above systems compared to *Webstrates* is the lack of support for real-time collaboration and multi-device distribution. By contrast, Croquet [28] which presents a collaborative 3D world that integrates its user and development environments, is closer in spirit to *Webstrates*. The main difference is that we use web-based technologies and do not use a 3D environment.

Several systems explore distributed user interfaces on the web. PlayByPlay [31] supports collaborative navigation; WebSplitter [12] distributes web pages across multiple devices for collaborative browsing; DireWolf [18] uses a widget-based design to share distributed pages; and Panelrama [32] divides web pages into panels distributed automatically across devices. These systems focus on distribution, but lack a general framework for shared content manipulation.

PolyChrome [1] is a web framework for creating collaborative and distributed web visualizations, using input-event redirection and synchronization to add collaboration support to single-user web pages. Heinrich et al. [13] take a different approach for creating multi-user web applications, using Operational Transformation [8] to synchronize the DOM. Unlike *Webstrates*, they synchronize only the part of the page corresponding to the domain object, e.g. an SVG element in an SVG editor, whereas *Webstrates* transparently synchronizes the entire DOM of any page.

Google Docs (*docs.google.com*) supports collaborative document editing and sharing of application extensions. However, it uses a traditional application model where extensions are limited to the provided document types and their APIs, whereas Webstrates relies on the general DOM API. Unlike Webstrates, Google Docs users must use the provided editor and the user interface is not a shareable object.

Many web frameworks, e.g., AngularJS (*angularjs.org*) or React (*facebook.github.io/react*), adapt the MVC pattern to web applications. All rely on synchronizing a JavaScript model with a DOM view. Combining these frameworks with real-time model synchronization through a shared database, e.g., Meteor (*meteor.com*) or Firebase (*firebase.com*), makes it possible to create real-time collaborative web applications. This approach differs fundamentally from *Webstrates*, which shares the DOM, that is, the pages themselves, rather than model objects. While MVC frameworks require application developers to manage the logic of updating the model when editing the view and updating the view as the model changes, *Webstrates* provides a dynamic medium that is inherently shared, without any explicit programming.

Edwards et al. [6] offer four approaches for creating infrastructures that better align user and system needs: surface, interface, intermediate and deep approaches. *Webstrates* focuses on a deep approach, built on a widely available technology. *Webstrates* also attempts to resolve the tension between encapsulating reusable components and supporting flexibility and tailorability [4] and to create an infrastructure where tools and documents can interoperate without prior knowledge, similar to Recombinant Computing [7].

Webstrates is also influenced by Instrumental Interaction [2], which separates the tools used for editing and manipulating content from the content itself. Unlike traditional applications that bundle the tools and the objects they edit, instrumental interaction promotes tools that are independent of

content and can be used in different contexts. With *Webstrates*, users can actually add and remove instruments at run-time. Scotty [5] breaks down some of these barriers by allowing developers to add new instruments to existing applications. VIGO [16] applies instrumental interaction to distributed, multi-surface environments, whereas Shared Substance [9] uses a shared data model to which clients can dynamically attach behavior, including via instruments. *Webstrates* use the DOM as shared data model for both content and instruments, providing a more unified approach.

WEBSTRATES CONCEPTS

Webstrates rely as much as possible upon existing web technologies. Individual pages (called webstrates) are served by a custom web server and accessed using regular URLs. They can be viewed in any modern desktop or mobile web browser (some features are only available in some browsers).

A webstrate is an HTML document that also contains CSS style sheets and JavaScript code (scripts). A webstrate is different from a regular web page, however. It includes a *Webstrates* client that sends any changes made to the DOM immediately to the *Webstrates* server, which stores it and sends it to any other client viewing that same webstrate (Fig. 2). All changes that affect the DOM are therefore synchronized in real time across all clients of the same webstrate. Changes to a webstrate can result from scripts embedded in the webstrate itself, from another webstrate that can access it through transclusion (see below), or from external modifications, e.g., through web browser developer tools.

The state of a web page that is not represented in the DOM is not shared by *Webstrates*. For example, scripts or style rules can be added without creating new elements in the DOM. This provides an escape mechanism to affect the presentation or behavior of a webstrate locally. Editing input fields also does not affect the DOM. When this is not the desired behavior, the webstrate must include code that reflects the edited value in the DOM, e.g., using an attribute of the input field.

Webstrates use *transclusion* as the main composition mechanism. Transclusion [22] is the inclusion of all or part of a document into another document simply by referencing it, in such a way that any change to the transcluded document is reflected in the document(s) transcluding it. In *Webstrates*, transclusion operates on entire webstrates. For example, a picture can be transcluded in a paper and in a slide deck. Any change to the figure is immediately visible in both the paper and the slide deck (Fig. 3). Transclusion is realized using the *iframe* element, which embeds a webpage and the *Webstrates* client inside another webpage.

Transclusion becomes particularly powerful when scripts embedded in the transcluding webstrate modify the content of the transcluded one and vice versa. For example, to edit an image, we create an editor webstrate containing the editing tools. These tools act on the content of the image transcluded inside the editor (Fig. 1b). Another approach is for the image webstrate to transclude the tools it needs.
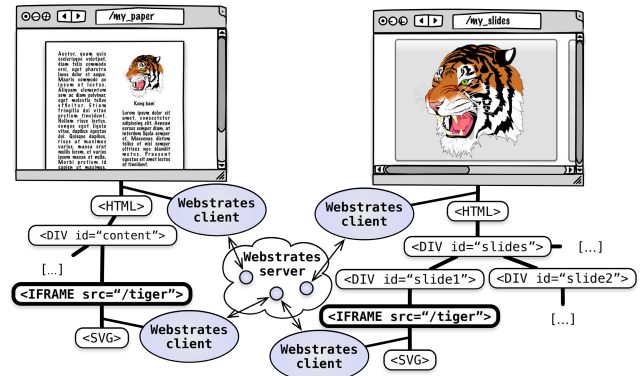

Figure 3: Two webstrates transclude the same figure. Changes to the figure appear immediately in both.

The advantage here is that the tools can be used on the image in context, wherever the image is embedded. This example shows how webstrates can act as documents (the figure), as applications (the image editor) that "open" the document via transclusion, or as a mix (the figure with embedded tools).

In order to assess *Webstrates*, we used it to collaboratively write the text and format the present paper (Fig. 1a,b) and to give presentations with active audience participation (Fig. 1c). We present our experiences in the form of two case studies, illustrated in the accompanying video, to demonstrate *Webstrates*' capabilities. The first presents a scenario showing collaboration between two co-authors who use different, personalized editors. The second presents a scenario showing the preparation and orchestration of a slideshow presentation with participants using multiple heterogeneous devices.

CASE STUDY 1: COLLABORATIVE PAPER AUTHORING

Alice, a graduate student in Europe, is co-authoring a research paper with Bob, a professor in the United States.

Personalized editors: Alice creates a new webstrate for the paper and loads it in her personalized editor webstrate, adds her notes and shares the document with Bob. She prefers editing in a plain text style, with a corresponding set of sophisticated layout and citation tools; whereas Bob uses a WYSIWYG editor with a print preview style that matches the final print layout. Alice uses an ACM style viewer webstrate on her tablet to see a live print preview as she writes.

Instrumental interaction: Alice has created a citation tool that works with the reference list in her bibliography webstrate (bibstrate). To add a new reference, she types its keyword and presses the *cite* button. If the key is found, the selected text is replaced with the citation (in the appropriate format), with a tooltip showing the full reference, which also appears in the References section.

Remote run-time interface extension: Bob wants to add his own references and asks Alice for help. She shares her tool by opening his editor on her computer and adds it to his toolbar; Bob can use it immediately.

Malleable user interfaces: Bob wants his toolbar to disappear except when the mouse hovers over that area. Alice

opens the code-editor webstrate and loads Bob's editor. She edits the style sheet of the editor and Bob sees the effect live.

Live transclusion of figures: Bob uses a stylus to sketch a figure in the drawing webstrate on his tablet. He adds the figure to the paper in the editor webstrate, and continues to make changes, which are updated live on both his laptop and the tablet. At the same time, Alice uses a more sophisticated vector graphics editor to clean up the lines.

How it works

Personalized editors: To create the webstrate for the paper, Alice copies a prototype webstrate consisting of a single editable element (using the *contentEditable* DOM attribute) and calls it *AliceBob2015*. Alice can now edit it directly and her changes are automatically saved. Alice types the name of the new webstrate in her personal editor (Fig. 1a, bottom), which includes her preferred style and tools, and the paper webstrate is transcluded into the editor. Bob creates his WYSIWYG editor by copying the ACM-provided webstrate and opening the paper webstrate (Fig. 1a, top).

The same paper is displayed with a different presentation in each editor. This heterogeneity seems incompatible with the notion of sharing as described so far: web documents contain their own style sheet and therefore sharing a document should also share its style. It is possible, however, to affect the style of a document in the browser without modifying its DOM by modifying the *document.styleSheets* object of the *iframe* that transcludes the document. Each editor holds the style to be applied to its document. When a document is loaded, the editor applies that style to the document's stylesheets object. Thus, the paper webstrate's style in the local browser context is changed without altering the DOM. The style applied to the document is itself in a separate webstrate that is transcluded by the editor, allowing, e.g., Alice to use the ACM style on her tablet.

In summary, the use of transclusion supports personalized editors while the injection of CSS rules supports different presentations of the same webstrate.

Instrumental interaction: We implement tools using the principles of instrumental interaction [2]. Each tool (or *instrument*) can operate on domain objects with certain well-defined properties. Instruments are not confined to the encompassing application, as in current systems. They are independent objects that can be moved, copied and shared. In *Webstrates*, the logic of each instrument resides in its own webstrate, which can therefore be shared using transclusion. For clarity, we refer to a webstrate that contains instruments as an *editor*. Editors typically contain a *toolbar*, i.e. an extensible panel that holds the instruments, and a *document* that can be edited by the instruments, which is usually a webstrate transcluded by the editor. Alice's editor features instruments for adding citations and comments, for tracking changes, and for applying styles.

An instrument webstrate contains parts that are shared by all editor webstrates that transclude it: the JavaScript code of its behavior and the HTML and CSS of its UI.
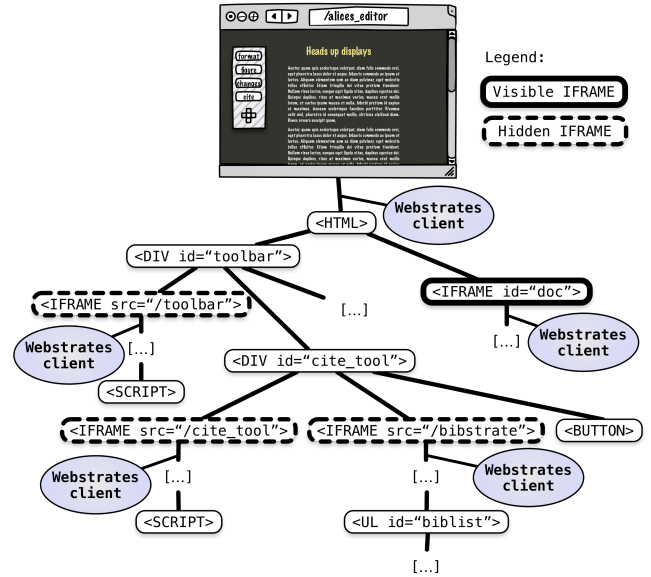

Figure 4: Composition of Alice's editor. The instantiation element of the toolbar and the citation tool are shown with the hidden webstrates they transclude.

An instrument also often needs state that belongs only to the hosting editor, e.g., the text color for an instrument that inserts comments or a toggle button for enabling and disabling change tracking. To add an instrument to an editor, we add both an *instantiation element* that contains these instance-specific elements and the transclusion of the instrument webstrate to a hidden *iframe*. When an instrument loads, it looks up its instantiation element in the parent webstrate and performs any necessary initialization, such as installing button listeners. Navigating to the instrument webstrate itself can provide documentation and a way to create an instantiation element.

The toolbar is itself an instrument that manages other instruments. Its instantiation element is the panel, which transcludes an instrument for adding and removing instruments (Fig. 4). The "+" button at the bottom of the toolbar opens a dialog for adding instruments; right-clicking on a tool opens a menu with a command to remove it.

The citation instrument (Fig. 4) includes an instantiation element that contains a hidden transcluded bibliography webstrate, or *bibstrate*, a button to open the bibstrate in a new window, and a button to insert a citation. Clicking the latter causes the citation instrument to search the bibstrate for a citation key matching the selected text and the document for an element with a *references* class. If not already present, it adds the reference to the references list. The citation instrument replaces the selected text with the proper citation, formatted according to the editor's style, e.g. "(Goodman, 1993)" in Alice's editor and "[11]" in Bob's.

The bibstrate is a list of citations in a structured DOM format. A separate instrument transforms BibTeX into this format to facilitate copy-paste from digital libraries.

In summary, instruments provide a flexible way to create and customize editors by decoupling documents from the tools used to edit them. *Webstrates* facilitate the creation and sharing of instruments, and their integration into editors.

Remote run-time interface extension: To share an instrument, one has to copy its serialized instantiation element and paste it in the target toolbar. When right-clicking a tool, the toolbar shows a menu with a command to show the HTML code of the instantiation element of that tool. Alice copies this code, clicks the "+" instrument in the toolbar of Bob's editor and pastes the code there. The toolbar instrument adds this code to the DOM and the scripts in the transcluded instrument are executed, making the citation instrument functional. Note that when Alice shares her citation instrument with Bob, his instrument will also share the same bibstrate as hers.

Automatic persistence and synchronization of webstrates, combined with the composition model of transclusion, enables functionality to be dynamically added and removed as simply as adding and removing content. These changes can also be made remotely, or by sending the HTML of the instantiation element over email, as in, e.g., Buttons [26].

Malleable user interfaces: To modify Bob's editor, Alice loads it into her code editor. The code editor transcludes the source webstrate in a hidden element, then populates a menu with the webstrate's scripts and stylesheets. Alice opens the stylesheet from Bob's editor and edits it to hide the toolbar by default and show it with the *:hover* selector. She then adds an animation attribute to make the toolbar fade in and out. Since stylesheets are automatically reinterpeted when changed, Bob can see the results of Alice's edits live. However, changes to scripts take effect only when a page reloads.

Such run-time collaborative tinkering is a key feature of malleable software and shareable dynamic media. It uses the same basic mechanisms as the manipulation of content, blurring the distinction between code and content, applications and documents, and development and use.

Live transclusion of figures: The drawing webstrate used by Bob transcludes a blank webstrate and lets Bob draw strokes (using Ploma: *plomaproject.tumblr.com*), which are turned into image elements in the figure webstrate. Bob uses an *add figure* instrument to transclude the figure in the paper and add an editable caption element. Alice's vector editor adds SVG elements to this webstrate, while Bob's drawing editor adds bitmap image elements. By compositing webstrates through transclusion, the figure updates live in the paper as multiple users edit it with different editors (Fig. 1b).

Summary
Webstrates offer a unified medium that blurs the distinction between content, computation and interaction. Alice and Bob interact with the same document via functionally and visually different webstrate editors. The same document (bibstrates) serves as both a document (collection of references) and a tool for managing those references. Leveraging the principles of transclusion and instrumental interaction, Alice and Bob can modify functionality at run-time, often without explicit programming. They can also incorporate more advanced programming into remote webstrates, live.

CASE STUDY 2: PREPARING AND GIVING A TALK
Alice and Bob must design the slides and create an interactive presentation with audience participation, controlled by the session chair.

Collaborative slideshow design: Alice opens a new slideshow webstrate with special tools for controlling slides. She wants to include several references, so she also adds her citation tool, as well as an annotation tool to add notes to each slide. She transcludes figures and graphs from the original paper and adds specific references. She sends the slideshow link to Bob and walks him through it using video chat. Alice navigates through the slides; Bob adds comments which Alice incorporates immediately.

Distributed slideshow presentation: At the conference, Alice meets Chuck, the session chair. He explains that audience members will see live copies of her slides on their devices, and be able to post questions during the talk. At the end, Chuck will select three that will appear in a webstrate in Alice's final slide. The session chair webstrate on Chuck's tablet lets him select the current presentation, see audience questions, and select those Alice should answer. During her presentation, Alice uses her tablet with her preferred presentation interface, which displays her notes and lets her control navigation. At the end of the talk, the questions selected by Chuck appear on her last slide, and she answers them.

How it works
Collaborative slideshow design: The slideshow editor is similar to the paper editor, except that it includes a different set of instruments (Fig. 5a). We use the Reveal presentation framework (*lab.hakim.se/reveal-js*) to represent the slideshow in HTML and CSS. Alice's citation instrument works, unmodified, as long as she includes reference sections at the end of the slideshow or on each slide. Figures can be transcluded into slides exactly as in the paper. Notes are added to slides as hidden elements. An instrument in the editor lets users view and edit the notes in a text area.

The Reveal framework uses the CSS class *present* to specify which slide is being shown. Since this class is added and removed from the DOM elements representing the root of each slide, any user viewing the slideshow webstrate will view the same slide, as well as the slide animations (which are implemented in CSS inlined in the slides) when changing the current slide. This is how Alice controls the presentation when showing it to Bob. The video chat between Alice and Bob is itself a webstrate that uses WebRTC. We use a WebRTC video chat service (*vline.com*) that can easily be embedded in a webstrate dedicated to Alice and Bob's chats.

Different types of content, such as papers and slides, usually require radically different editors and copying and pasting content. With *Webstrates* they are represented in the same medium and can share content as well as instruments. *Webstrates* also work with popular web frameworks and advanced features such as the recent WebRTC protocols.
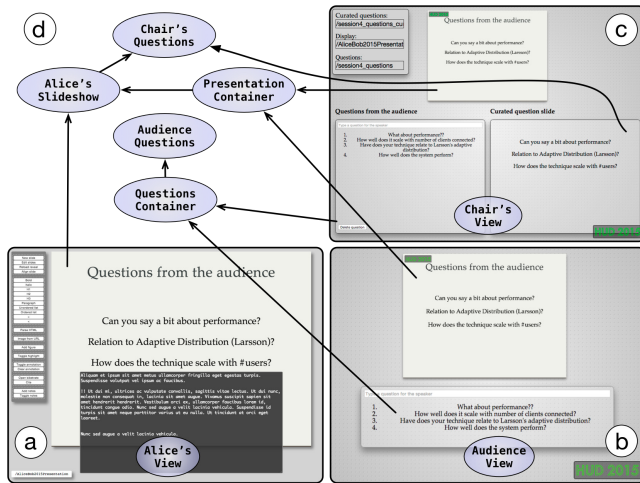
Figure 5: (a) Alice's presenter view, (b) Audience view, (c) Session chair view, and (d) Architecture of these three linked presentation webstrates. Ovals indicate webstrates and arrows indicate transclusion.

Distributed slideshow presentation: The projector view (Fig. 1c) is a simple container webstrate that transcludes a slideshow. The audience view (Fig. 5b & 1c) transcludes the slideshow container together with a container for a questions webstrate. The questions webstrate contains a form for submitting questions. Questions added to the webstrate are visible to the rest of the audience and the session chair. The chair's view transcludes the slideshow and the question containers together with a webstrate where the chair can copy/paste questions and edit them (Fig. 5c). The chair's edited questions are transcluded on Alice's last slide. The chair can control which webstrates are loaded in the containers using a form. Figure 5d gives an overview of the architecture.

To control the presentation, Alice uses a webstrate that transcludes her slideshow, with a style that provides navigation tools and makes her presenter's notes visible. We have also implemented the automatic transclusion of the session webstrate using WiFi proximity detection [17] so that the audience can use a single webstrate throughout the conference and see the slides and questions for the session they are attending. The webstrate polls a location service and maps the current location and time to a session webstrate, which is automatically loaded into the participant's container. Here, we take advantage of the ability to change the address of the *iframe* transcluding the session webstrate without changing the DOM, so that all participants can share the same proximity detection webstrate.

Webstrates can be used to orchestrate a complex distributed and collaborative session in a simple way, by combining existing webstrates and creating containers and simple tools to configure the session. *Webstrates* can also seamlessly integrate ubicomp services such as location information.

Summary
Webstrates offer a medium that breaks down the barriers between documents and applications and facilitates tool reuse. The inherent sharing mechanism allows users to orchestrate complex collaborative and distributed situations involving multiple users with different roles and devices.

IMPLEMENTATION
Webstrates consists of a server and a client. The server is based on Node.js and consists of ~300 SLOC of CoffeeScript (excluding third-party libraries). The server injects the client into the web browser. It consists of ~800 SLOC of Coffee-Script (excluding third-party libraries). The code is available at *http://www.webstrates.net*.

Automatic DOM synchronization
To synchronize the DOM between clients, we use ShareJS (*sharejs.org*), an open source library implementing Operational Transformation [8] based on a Jupiter client/server [23]. ShareJS supports plain text and JSON. We transform HTML documents (with inlined CSS and JavaScript) into JsonML (*jsonml.org*). Our server stores the documents along with their operation log in a MongoDB database.

When a browser fetches a webstrate, e.g. http://webstrates.net/my_webstrate, the server sends the *Webstrates* client and the id of the document ('my_webstrate'). The client then retrieves the JsonML document with the given id ('my_webstrate') from the server, converts it to HTML, and loads it into the browser window. (Note that the *Webstrates* client is still loaded in the browser window.)

The client uses the MutationObserver DOM API to observe changes to the local DOM including inlined scripts and styles. When a mutation is observed, the mutation is translated into a JSON operation on the ShareJS document and sent to the server, which records it and propagates it to the other clients. Conversely, when a JSON operation is received from the server, it is translated into a manipulation of the DOM. Since the mutations generated by a MutationObserver arrive asynchronously with only relative location information, e.g., a node was added to a parent next to this immediate previous sibling, we maintain an intermediate representation of the DOM in order to compute absolute paths when generating JSON operations for each individual mutation.

***Webstrates* API**
The *Webstrates* client adds two new events to the standard DOM API: it fires an event on a window when it has finished loading the content of a webstrate, and when the content of a text node in the DOM is replaced. In the latter case, the event contains the difference with its previous value. Apart from these two events, programming with *Webstrates* relies exclusively on the browser's DOM API. Synchronization between clients and the persistence of the state are completely transparent. In particular, most existing libraries used by web developers can be used to create and manipulate rich content.

New webstrates are created either by sending a request to the server with an id that does not exist, or by copying an existing webstrate: The root URL of the server (e.g. http://webstrates.net/) is a webstrate that lets users create a copy from a list of common webstrates or from a named webstrate.

Authentication

Users authenticate with the server using external providers, e.g. Github or Twitter. The server stores the access rights for a webstrate in an attribute of its *html* tag. A more refined authentication and access rights model is left for future work.

Transclusion

Transclusion uses an *iframe* element, which is a standard HTML element designed to embed a page within the surrounding page. By loading a webstrate in the *iframe* element, we achieve exactly the effects of transclusion: changes to the transcluded webstrate that are made, e.g., in another client, propagate and show up immediately (Fig. 3). For security reasons, the browser insulates the content of *iframes* from their surrounding page. However, when both pages are served by the same server, each can access and modify the other, opening the way to the types of composition mentioned earlier and illustrated in the examples below.

Performance

Performance is excellent, even over transatlantic distances, due to the use of Websockets and asynchronous handling and storage of operations on the server. This article was written entirely with *Webstrates* and the co-authors always experienced interactive response times. Each webstrate loaded into the browser has its own instance of the client and socket connection to the server, which could be optimized. Also, the server keeps the entire log of operations on each webstrate, which could be culled. To ensure fast loading, the server stores a snapshot of a recent version of each webstrate.

The most computationally intensive parts of the *Webstrates* client are computing the absolute path of an observed mutation and the difference between the old and new values of a text node. With our intermediate representation of the DOM, we can compute the absolute path of a mutation in a time proportional to the height of the DOM tree. To compute the difference between two text nodes we use a third party library based on Myers' algorithm [21] with a complexity of $O(ND)$, where N is the combined length of the two strings and D is the size of their minimum edit script (diff).

DISCUSSION

Working with *webstrates*

Creating webstrates is very similar to traditional web development with HTML, CSS, JavaScript and popular libraries such as jQuery. It is very different, however, from developing web *applications*, which require frameworks that use complex logic to render content from a back-end database into HTML and interpret user input into database queries. With *Webstrates*, the medium is the model: any change to the content of a page is shared among clients and stored in the server. This requires some adaptation, and a different set of design patterns than for traditional, MVC-based applications.

In our own experience, creating webstrates is surprisingly easy. The ability to work directly in the browser, using developer tools to make changes and seeing those changes synchronized immediately gives the sensation of working within a live medium. In some cases however, achieving the desired effect is more complicated. This is usually the case when trying to violate a fundamental tenet of *Webstrates*: *a webstrate shares the DOM, the whole DOM and nothing but the DOM*. Problems typically come from either wanting to share information that is not represented in the DOM or *not* wanting to share information that is in the DOM.

The first problem can be addressed by simply adding the information to the DOM. For example, to share the content of a text field, we use two listeners to synchronize the value of the field with an attribute that we add to the field. Since attribute values are synchronized by *Webstrates*, we effectively share the value of the field in real time.

The second problem, not sharing information already in the DOM, is more complex, since we cannot simply remove that information. A webstrate is identical for all users, and cannot directly contain different tools for different users. The pattern we use most often is a container webstrate that contains the custom elements (here, the tools) and also transcludes the webstrate to be customized (here, the content). This way, different users can use different containers, tailored to their needs. However, this solution does not cover cases where custom elements must be embedded directly in the document, such as personal comments on the text. In such cases, we do include the content in the DOM, and use local style rules injected in the document to hide unwanted content.

Another important design pattern is the use of instruments to separate content from the tools used to edit it. As explained in the first case study, we use an instantiation element that transcludes the instrument. To ensure that instruments work in a variety of contexts, we query the DOM to find the target of the instrument. This often involves crossing *iframe* boundaries to reach a container or a transcluded webstrate. This has proven to be a source of difficulty, especially when using existing web libraries, because the scripts must be evaluated in the target *iframe*. For example, while Reveal.js worked out of the box for the slideshow, CodeMirror (*codemirror.net*), used for our code editor, does not play well with our persistent DOM. We therefore run it in a separate *iframe* and synchronize its state explicitly. While this loses some of the benefits of *Webstrates*, it demonstrates that it is possible to integrate otherwise incompatible third-party libraries.

The last pattern we use is a controller webstrate, reminiscent of MVC, where the controller transcludes a "view" webstrate that it updates as needed, e.g. by monitoring an external source as in the tangible clock example below. However, "view" webstrates are full-fledged webstrates and can be shared as any other webstrate.

Simplicity and generative power

We present three working prototypes that illustrate the simplicity and generative power of *Webstrates,* beyond document editing and presentation. Each shows how to reframe an existing system in terms of shareable dynamic media and how our approach suggests ideas for novel capabilities.

Shared window manager

We explored how webstrates could replace an entire desktop

by implementing a simple window manager webstrate where each window transcludes a webstrate or a regular web page. The user can move, resize and scale each window, and interact with its content as usual. An interesting added benefit is that sharing the window manager webstrate provides a simple and efficient screen-sharing solution.

We ran this window manager on an interactive wall-sized display with 75 LCD tiles powered by a cluster of 10 computers (Fig. 6a). We created a different container webstrate for each computer that transcludes the window manager webstrates, showing only the part corresponding to the tiles it controls. Performance is excellent, as seen in the video.

Communication appliance
MarkerClock [25] displays the activity of several remote users over time (Fig. 6b). We recreated it with a controller webstrate, which transcludes a clock webstrate that displays the clock face, and an activity webstrate, which captures user activity. The controller updates the hands of the clock, monitors user activity by analyzing the video feed from a camera, and adds marks to the activity webstrate.

Tangible world clock
We designed a tangible LEGO world clock controlled by a LEGO Mindstorm EV3 and an iPod Touch (Fig. 6c). The user sets the time by turning the hands of the LEGO clock and changes the timezone by tapping the iPod display. The iPod runs a controller webstrate that transcludes a clock webstrate. The controller communicates with the EV3 brick over a websocket. When it receives events from the brick, it updates the time on the clock webstrate. When the timezone is changed, it sends the new time to the brick. Opening the clock webstrate on another device creates a remote control and display for the physical clock.
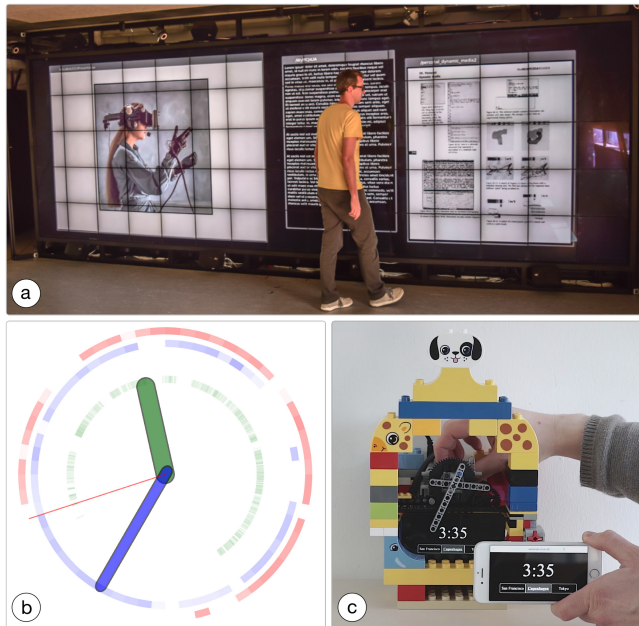
Figure 6: (a) Window manager webstrate running on a 10-computer wall-sized display. (b) Marker Clock activity tracker. (c) Tangible world-clock.

Systems-oriented evaluation criteria
A user interface software tool should have a low threshold —be easy to learn—and a high ceiling—support advanced users [20]. Or, as Alan Kay put it, "simple things should be simple, complex things should be possible." The threshold for learning *Webstrates* is limited to basic knowledge of HTML, CSS and JavaScript: there is no new API to learn, and sharing is free. Conversely, the ceiling of *Webstrates* is that of the web, with the ability to distribute and synchronize content in real time. However, what is rendered outside the DOM, e.g., on the HTML5 canvas or with WebGL, or stored in a JavaScript model or on a server is not synchronized and requires ad hoc solutions. The case studies and examples above demonstrate that complex scenarios involving multiple users and devices can be implemented relatively simply.

In order to assess whether "simple things are simple," we implemented the to-do list benchmark from *todomvc.com* with *Webstrates*. This is a simple to-do list where users can add, remove, check and filter to-do items. Our implementation, which uses jQuery, is only 53 logical lines of code (lloc), compared to 110 lloc with Meteor and 90 lloc with Firebase. The code is not only short, but also simple: it is the same code that one would write to create a single-user to-do list without persistence. The only exception is an attribute on a checkbox element and a mutation observer used to synchronize its run-time state, which is not stored in the DOM.

For our case studies, the size of JavaScript code is 932 lloc for case study 1 (toolbar: 75 lloc; citation tool: 68 lloc; figure tool: 68 lloc; sketching tool: 123 lloc; code editor: 184 lloc) and 123 lloc for case study 2 (slideshow instrument: 83 lloc; session chair webstrate: 33 lloc). The window manager is 117 lloc, MarkerClock 44 lloc and the LEGO clock 148 lloc.

Olsen [24] introduced several values of a good user interface toolkit, including reduced viscosity through flexibility, expressive leverage and expressive match. Flexibility is the ability to make rapid changes that can quickly be evaluated by users. *Webstrates* are highly flexible: a developer can tinker with a user's interface remotely at run-time with immediate response or, at worst, after reloading the page. Expressive leverage is "where the designer can accomplish more by expressing less." *Webstrates* achieves expressive leverage by making real-time sharing a fundamental feature of the medium rather than requiring specialized frameworks as in current web applications. For example, the list of questions in the slideshow case study is implemented without having to think about synchronization, database back-end, etc.

Expressive match is "an estimate of how close the means for expressing design choices are to the problem being solved." By providing an environment where both use and development take place in the browser, *Webstrates* makes development more direct. Because sharing applies to the DOM itself, rather than to invisible model objects, the environment is more transparent and the solutions are expressed in terms of what is shared by whom. Also, *Webstrates* are compatible with many web frameworks, which helps increase expressive match for the issues that they do not address. Finally, *Web-*

strates provide high expressive match to the users themselves by letting them configure their environment to suit their needs. For example, one user may prefer a visual color wheel to pick a color while another prefers entering a hex string. With *Webstrates*, both can work on the same document with their own tools and are not limited to those bundled with traditional applications.

Limitations

Although *Webstrates* is fully functional, the current implementation has some technical limitations and lacks certain features. First, the Operational Transformation algorithm does not recognize move operations, which are interpreted as delete and insert. This causes problems when a user is editing a section that is moved by another user. We are considering other algorithms, such as CRDTs [27]. Second, all communication is unencrypted and our authentication and access rights are overly simple. Third, while performance has been satisfactory so far, the system could be optimized. In particular, all webstrates in a page should share the same client and websocket connection. We are also considering a peer-to-peer architecture so as to be less dependent on a central server. Finally, a document manager webstrate would help users organize their webstrates, and presence indicators would improve awareness of other users.

CONCLUSIONS AND FUTURE WORK

Webstrates offer a novel approach for creating shareable dynamic media that blur the distinction between documents and applications. Our vision requires software to be *malleable* by users, so they can appropriate documents and tools to meet individual needs; *shareable* among users, so they can collaborate on multiple aspects of the media; and *distributable* across heterogeneous devices and platforms. We build upon earlier visions of the Dynabook and ubiquitous computing, but with today's technology ecosystem, and describe how relatively minor changes to current web technology can bring our vision closer to reality.

We introduce *Webstrates*, an exploratory platform that turns web pages into webstrates that embody content, computation and interaction. Webstrates act as documents, tools or applications, depending upon context of use. They are shareable among users and changes are automatically synchronized among clients and made persistent. We support live composition of diverse media into a single, shared document, using transclusion. Users can create and modify their own tools at run-time, using instrumental interaction. As a proof of concept, the co-authors wrote this paper using personalized editor webstrates matching their individual work styles and technical skills. We demonstrated the power and simplicity of *Webstrates* by creating a shared window manager, a communication appliance, and a tangible interactive clock. Future work will explore how *Webstrates* can support diverse computer-mediated activities, especially to support creative and scientific activities.

ACKNOWLEDGEMENTS

REFERENCES

1. Badam, S.K. and Elmqvist, N. (2014) PolyChrome: A Cross-Device Framework for Collaborative Web Visualization. In *Proc. Interactive Tabletops and Surfaces (ITS '14)*. ACM. 109–118

2. Beaudouin-Lafon, M. (2000) Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proc. Human Factors in Computing Systems (CHI'00)*. ACM. 446–453

3. diSessa, A.A. and Abelson, H. (1986) Boxer: A Reconstructible Computational Medium. In *Commun. ACM*. **29**(9). 859–868

4. Dourish, P. and Edwards, W.K. (2000) A Tale of Two Toolkits: Relating Infrastructure and Use in Flexible CSCW Toolkits. In *Jal. CSCW*. **9**. Springer. 33–51

5. Eagan, J.R., Beaudouin-Lafon, M. and Mackay, W. (2011) Cracking the cocoa nut: user interface programming at runtime. In *Proc. User Interface Software and Technology (UIST'11)*. ACM. 225–234

6. Edwards, W.K., Newman, M.W. and Poole, E.S. (2010) The Infrastructure Problem in HCI. In *Proc. Human Factors in Computing Systems (CHI'10)*. ACM. 423–432

7. Edwards, W.K., Newman, M.W., Sedivy, J.Z. and Smith, T.F. (2009) Experiences with Recombinant Computing: Exploring Ad Hoc Interoperability in Evolving Digital Networks. In *TOCHI*. **16**(1). ACM. 3:1–3:44

8. Ellis, C.A. and Gibbs, S.J. (1989) Concurrency control in groupware systems. In *SIGMOD*. **18**(2). ACM. 399–407

9. Gjerlufsen, T., Klokmose, C.N., Eagan, J., Pillias, C. and Beaudouin-Lafon, M. (2011) Shared substance: developing flexible multi-surface applications. In *Proc. Human Factors in Computing Systems (CHI'11)*. ACM. 3383–3392

10. Goldberg, A. and Robson, D. (1983) Smalltalk-80: the language and its implementation. Addison-Wesley.

11. Goodman, D. (1993) The complete HyperCard 2.2 handbook. Bantam books.

12. Han, R., Perret, V. and Naghshineh, M. (2000) WebSplitter: a unified XML framework for multi-device collaborative Web browsing. In *Proc. Computer Supported Cooperative Work (CSCW'00)*. ACM. 221–230

13. Heinrich, M., Lehmann, F., Springer, T. and Gaedke, M. (2012) Exploiting single-user web applications for shared editing: a generic transformation approach. In *Proc. World Wide Web (WWW'12)*. ACM. 1057–1066

14. Kay, A. (1972) A personal computer for children of all ages. In *Proc. ACM Annual Conference V1.1*. ACM.

15. Kay, A. and Goldberg, A. (1977) Personal dynamic media. In *Computer*. **10**(3). IEEE. 31–41

16. Klokmose, C.N. and Beaudouin-Lafon, M. (2009) VIGO:

instrumental interaction in multi-surface environments. In *Proc. Human Factors in Computing Systems (CHI´09)*. ACM. 869–878

17. Klokmose, C.N., Korn, M. and Blunck, H. (2014) WiFi proximity detection in mobile web applications. In *Proc. Engineering Interactive Computing Systems (EICS´14)*. ACM. 123–128

18. Kovachev, D., Renzel, D., Nicolaescu, P. and Klamma, R. (2013) Direwolf-distributing and migrating user interfaces for widget-based web applications. In *Web Engineering*. Springer. 99–113

19. Krahn, R., Ingalls, D., Hirschfeld, R., Lincke, J. and Palacz, K. (2009) Lively Wiki a development environment for creating and sharing active web content. In *Proc. Wikis and Open Collaboration*. ACM. 9

20. Myers, B., Hudson, S.E. and Pausch, R. (2000) Past, present, and future of user interface software tools. In *ACM TOCHI*. **7**(1). ACM. 3–28

21. Myers, E.W. (1986) An O(ND) difference algorithm and its variations. In *Algorithmica*. **1**(1-4). Springer. 251–266

22. Nelson, T.H. (1995) The Heart of Connection: Hypermedia Unified by Transclusion. In *Commun. ACM*. **38**(8). ACM. 31–33

23. Nichols, D.A., Curtis, P., Dixon, M. and Lamping, J. (1995) High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. In *Proc. User Interface Software and Technology (UIST´95)*. ACM. 111–120

24. Olsen Jr, D.R. (2007) Evaluating user interface systems research. In *Proc. User Interface Software and Technology (UIST´07)*. ACM. 251–258

25. Riche, Y. and Mackay, W. (2010) PeerCare: supporting awareness of rhythms and routines for better aging in place. In *Proc. Computer Supported Cooperative Work (CSCW´10)*. **19**(1). Springer. 73–104

26. Robertson, G.G., Henderson Jr, D.A. and Card, S.K. (1991) Buttons as first class objects on an X desktop. In *Proc. User Interface Software and Technology (UIST´91)*. ACM. 35–44

27. Shapiro, M., Preguiça, N., Baquero, C. and Zawirski, M. (2011) Conflict-free Replicated Data Types. In *Proc. Stabilization, Safety, and Security of Distributed Systems (SSS´11)*. Springer-Verlag. 386–400

28. Smith, D.A., Kay, A., Raab, A. and Reed, D.P. (2003) Croquet-a collaboration system architecture. In *Proc. Creating, Connecting and Collaborating Through Computing (C5´03)*. IEEE. 2–9

29. Taivalsaari, A., Mikkonen, T., Ingalls, D. and Palacz, K. (2008) Web Browser As an Application Platform: The Lively Kernel Experience. In *SMLI TR-2008-175*. Sun Microsystems, Inc.

30. Weiser, M. (1991) The computer for the 21st century. In *Scientific American*. **265**(3). Nature Publishing. 94–104

31. Wiltse, H. and Nichols, J. (2009) PlayByPlay: collaborative web browsing for desktop and mobile devices. In *Proc. Human Factors in Computing Systems (CHI´09)*. ACM. 1781–1790

32. Yang, J. and Wigdor, D. (2014) Panelrama: enabling easy specification of cross-device web applications. In *Proc. Human Factors in Computing Systems (CHI´14)*. ACM. 2783–2792