Deploying Stateful Web Components on Multiple Devices with Liquid.js for Polymer

Andrea Gallidabino
Faculty of Informatics
University of Lugano (USI)
CH-6900 Lugano, Switzerland
andrea.gallidabino@usi.ch

Cesare Pautasso
Faculty of Informatics
University of Lugano (USI)
CH-6900 Lugano, Switzerland
c.pautasso@ieee.org

***Abstract*—Nowadays, the average users owns two or more Web-enabled devices (smart phones, personal computers and tablets), while more are coming: watches, cars, glasses. While responsive Web applications can adapt to the specific device (e.g., screen size or input method) on which they are deployed, they only provide limited support when one user connects from more than one device at the same time. In this paper we present Liquid.js for Polymer, a framework whose goal is to support developers that need to build liquid Web applications taking full advantage of multiple heterogeneous devices. It extends emerging Web components standards to enable the liquid user experience, whereby any device can be used sequentially or concurrently with applications that can roam from one device to another with no effort required by users controlling them. This way, users do not need to stop and resume their work on their Web application as they switch devices. Developers using Liquid.js for Polymer do not need to worry about the underlying connectivity, pairing and synchronization issues of their stateful Web components.**

***Keywords*-Web Components, Liquid Software, Liquid Web Applications, Stateful Web Components**

I. Introduction

On average a user owns two or more Web-enabled devices [9], ranging from more common smart phones, tablets and personal computers, to newest technologies such as smart watches and wrist-bracelets. In the near future the set of programmable Web enabled devices is likely to grow even more [11]. While responsive Web applications can adapt to each device running them [17] and every Web application is designed to scale to handle a large number of concurrent users, currently the architecture of most Web applications is not designed to withstand the interactions of one user accessing their application from multiple heterogeneous devices.

Liquid is a metaphor which characterizes software designed to be seamlessly deployed on multiple heterogeneous devices [1]. The metaphor was inspired by the ability of a liquid to adapt its own shape to the one of the container holding it. More precisely Liquid software is able to flow (e.g. migrate) between devices without losing the focus and the state of the application ran by users [16]. A software is considered liquid whenever its layers [18] can adapt to take full advantage of every device on which they are deployed on.

The liquid software metaphor was originally proposed in the context of active networks research in 1996 [12]. The platform Joust [13] tied liquid software with code mobility [6]. Other similar metaphors appeared in literature, like the concept of *Fluid computing* [2] by Bourges-Waldegg et al. In the metaphor the state of an application is seen as a *fluid* and the research takes into account data synchronization and replication in multi-device applications. More recently, the Liquid Software Manifesto [23] introduced the concept of *Liquid User Experience*.

In this context, we propose to apply the liquid metaphor to stateful software components and present our design of the Liquid.js for Polymer framework, targeting Web components, an emerging technology for building component-based Web applications. We discuss our solution for the declarative and explicit representation and synchronization of the runtime state of a component, so that developers can have full and fine-grained control over where and when the state of components is persistently stored. In previous works [7] we have evaluated the framework expressiveness with realistic application examples and demonstrated how developers can easily inject the liquid behavior into standard Web components so that the user can dynamically migrate or clone them on different devices.

Liquid.js aims to help developers build their own Liquid component-based Web applications able to run seamlessly on multiple heterogeneous devices. The framework will also provide the tools to support the *Liquid User Experience*, by offering a set of primitives (migration, forking, cloning) allowing developers to build Liquid software without dealing with the underlying complexity of developing their own real-time state synchronization infrastructure [8].

II. Related Work

Our work is inspired by Grundy et. al. [10], who proposed a thin-client approach where collaborative applications are distributed across multiple devices. Their study however did not take into consideration liquid behavior and the corresponding state synchronization.

The growth in complexity of Rich Internet Applications has been surveyed by Casteleyn et al. [3], who observed that larger and larger amounts of state are accumulated during normal usage sessions, making it challenging to migrate modern Web applications across different devices. Likewise, it is also difficult to synchronize such shared state among multiple Web

browsers that access the same Web application concurrently on behalf of the same (or different) user.

Together with the TUT Liquid software group, we applied the concept of liquid software to Web applications by providing usage scenarios and possible solutions to take advantage of existing or emerging Web technologies for creating Liquid Web applications [18].

Component-based Web applications can also be built as mashups of Widgets, encapsulating user interface components, their behavior and state. Chudnovskyy et al. [4] presents a solution to develop a (semi) automatic system that creates the inter-communicating mesh between all the widgets embedded in a Web page. The same authors [5] discuss in more details the challenges of inter-widget communication in widget-based mashups and composite Web applications, in particular: *awareness*, the ability of the user to know which widgets are connected; and *control*, the ability of the user to change the mesh of connections among the widgets. Liquid.js takes a very similar approach by using *Web components*, which provide reusable user interface widgets using standard Web technologies but targets the development of Web applications that can be distributed over multiple Web browsers.

Outside the Web browser, there exist many attempts to enable applications to flow between mobile devices or mobile and desktop operating systems. One of most notable examples are *continuity* and *handoff* by Apple. With continuity it is possible to migrate a call from one device to another, while with Handoff it is possible to roam from device to device while holding the work session that was initially created, for example while composing an email. In the Apple implementation [20], devices discover one another via bluetooth and the synchronization mechanism of the application state is provided by the centralized iCloud backend.

Liquid applications with simultaneous screening require to deal with replicated data synchronization issues, which have been studied in the database community for many years [14]. On the Web, given the heterogeneity of the devices (i.e., the amount of storage may vary) and their frequent disconnected state (e.g., due to battery or network connectivity problems), some of the assumptions of classical data replication mechanisms may need to be revisited.

III. The Liquid.js for Polymer framework

The Liquid.js framework targets the development of Web applications that require support for sequential and simultaneous screening across multiple devices owned by the same user, and can be generalized to collaborative scenarios involving multiple devices owned by multiple users. The framework enables the creation of transparently decentralized Web applications that need to operate on a shared decentralized state. The assumption is that these applications are developed using Web components, which provide the necessary level of granularity to structure the application user interface and its state. Liquid.js takes care of the state synchronization by using Yjs [19], a concurrency control and conflict resolution framework.

The Liquid.js framework is built for developers whose goal is to easily create applications able to run on multiple devices transparently implementing all the features of Liquid software. The framework supports three scenarios: 1) **sequential screening**, a single user owns two or more different devices, and runs an application at different times on different devices; 2) **simultaneous screening**, a single user owns two or more different devices, on which he runs an application simultaneously on more than one device; 3) **collaboration**: a set of users collaborate using on the same application running on all the devices they own. Liquid.js provides developers with the following abstractions (Figure 1a):

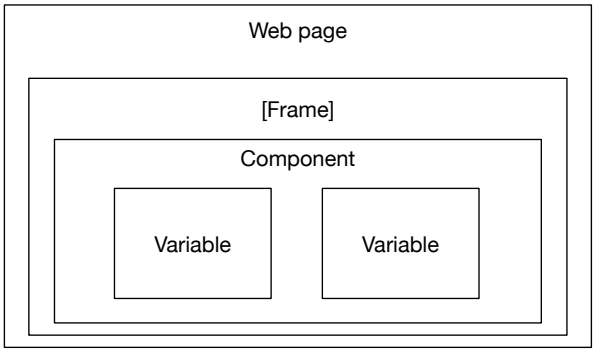
- **Liquid component**: a piece of executable code defined by both JavaScript and HTML. The Liquid component encapsulates both the state and the Liquid behavior, which is the core module that provides the Liquid API to the component.
- **Liquid frame**: The Liquid frame is an optional UI container surrounding a Liquid component that enables liquid user experience by providing support for the interactions of the user with the Liquid.js library. It provides a default user interface visualization of migration, cloning, and destruction operations, as well as feedback on the available devices. Component developers can choose to expose their own liquid controls implemented directly using the Liquid API. In this case, they can hide the default Liquid frame, which is primarily meant to surround legacy Web components to which the liquid behavior has been attached.
- **Liquid variable**: The Liquid variable is the smallest part of state contained in a Liquid component. The Liquid variable is defined by a *name*, a *value*, and some *permissions*. The state of a Liquid component is defined by all Liquid variables it contains.

Figure 1b shows one possible component created in Liquid.js. The Liquid Webcam component shares pictures taken from the Webcam feed with all other paired instances of the same component. This component defines two Liquid variables to store the pictures to be shared with all other webcam components and to display the webcam video feed. The Liquid frame, the dark bar above the component, includes buttons and icons than enhance the liquid user experience. This frame lets the users interact with the Liquid.js API and trigger the instantiation, migration, cloning or destruction of a particular Liquid component.

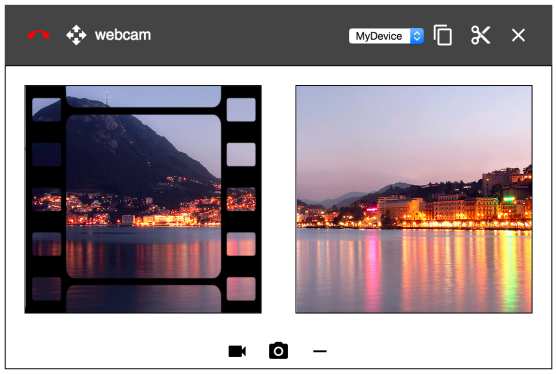
A. Liquid components

While many Web applications are built following the model-view-controller pattern [15], we are concerned not only about the synchronization of the data model of the application, but also about the state of individual UI components, which needs to be properly migrated and synchronized when such components begin to flow between multiple devices.

Compared to other HTML5-based solutions [22], we decided to use a different approach: we based our design on a component-based architecture used to enhance liquidity in all layers of an application. The Liquid components are built on top of the Web Components standard using the



(a) Liquid Web application structure



(b) Example Liquid Web application with the webcam component. The frame controls the liquid user experience, the variables store the image snapshot and the webcam video feed.

Fig. 1: Liquid.js abstractions (Frame, Component, Variable) and example liquid Web application

Polymer syntax. This way we can instantiate multiple stateful Liquid components isolated from each other. In addition, with Liquid.js developers can explicitly declare which part of the component state needs to be synchronized.

We take a component-based approach because our goal is to have fine-grained control over which parts of the Web applications should run on each device. The decomposition allows fine-grained control over the dynamic deployment configuration of the Web application as components are instantiated and migrated from a device to another in response of user commands. By decomposing the UI we also can reduce the size of the state that is tied to each component. Thus, the detection of changes in the state of the application and their propagation across other devices, on which the corresponding components have been cloned, can be more efficient.

The Liquid components are dynamically loaded in the Web application by using the *HTML imports* API. A Liquid component encapsulates *behaviors* (JavaScript) and *UI* (HTML). Whenever a Liquid component is needed, it is dynamically loaded in the page through the *HTML imports*. Figure 3 shows the internal structure of the Liquid component. A liquid component is defined by its *Web Component*, which is the expected *solid* functionality of the component implemented by the developer, and the *Liquid behavior*, a Polymer behavior which allows automatic state synchronization and data storage between multiple instances of the component provided by Liquid.js.

The Liquid behavior takes care of propagating and synchronizing changes of the state in all component instances transparently. The developer does not have to worry about the location and the number of instances running, nor keep track of the set of devices that are connected to the application on behalf of the user. The state synchronization messaging with other Liquid components is also managed transparently. By exploiting the WebRTC standard protocol, the Liquid components exchange peer-to-peer messages between them, freeing the server from relaying WebSockets messages whenever applicable, e.g. if the WebRTC Datachannels are available, otherwise the communication will relay on the *Liquid Server*. The Liquid Server is also used for device discovery and pairing whenever a new WebRTC communication must be established, moreover it can be used as storing repository for the Liquid components that need be imported and loaded in the Web page.

The liquid component lifecycle API makes it possible to create new components and move them from a device to another. The API exposes the following methods:

- createComponent(componentType [, deviceID])
- pairComponents(componentID$_1$, componentID$_2$)
- unpairComponents(componentID$_1$, componentID$_2$)
- removeComponent(componentID)
- moveComponent(componentID, deviceID)
- cloneComponent(componentID, deviceID)
- forkComponent(componentID, deviceID)

B. Liquid variables

The state of a component is decomposed into one or more Liquid variables. A variable is identified by its name and its current value can hold any JSON-serializable JavaScript data type. Values of Liquid variables are automatically synchronized among paired component instances, according to the permissions associated to each variable.

While each Liquid component instance holds the current synchronized state of a Liquid variable, the Liquid.js framework may choose to replicate them elsewhere. This approach ensures that the value of the state is as close as possible to the source using it, while the data synchronization mechanism is as close as possible to all parties participating in the synchronization.

The Liquid State Storage API helps developers declare which state variables should be managed by Liquid.js so that their values get automatically synchronized. The API exposes the following three methods:

- registerVariable(name, init[, permissions])
- variableChange(name, value)
- pairVariable(compID$_1$, name, compID$_2$, name$_2$)

C. Permissions

The permissions affect the synchronization of Liquid variables. They define how the state of a variable can be changed

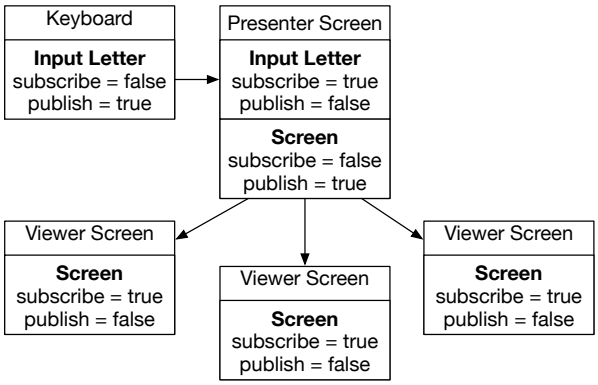


Fig. 2: Example permissions

and propagated among paired components. There are two Boolean permissions: 1) **publish**; 2) **subscribe**. The publish permission defines if a component is allowed to propagate changes in the state of a variable to other paired variables. The subscribe permission defines if a component is allowed to accept changes from another paired component.

The publish and subscribe permissions are not only used to define the flow of the data among components, but they also prevent malicious or glitchy components to access data they should not. Granting control to both parties of the communication is important to avoid unwanted interactions since liquid components run on a Web Browser and permissions can be unilaterally changed by malicious users.

These permissions make it possible to create useful propagation patterns. Figure 2 shows one possible application composed by three different Liquid components: *Presenter Screen* which shows in a text-area all the letters input by a keyboard; the *Keyboard* which is used to input letters; and the *Viewer Screen* which shows the same text shown inside the *Presenter Screen*. In this application *Presenter Screen* pairs itself with a *Keyboard* component. Since they both define an *Input Letter* variable they share it. The keyboard publishes the input letter in the presenter screen, which allows subscription to the keyboard. Similarly the *Presenter Screen* pairs with many *Viewer Screens*. The viewer screens allows subscription to the variable *Screen*, the presenter screen publishes any new value to all viewers.

D. Liquid Storage

The decision of where a liquid variable is stored is affected by the following dimensions:

1) **Sharing Policy**: The sharing policy defines how many components a variable is shared with. The possible values of the sharing policy are:

- **Global**: a global variable is shared with all instances of all components **automatically**. It is not necessary to pair them explicitly.
- **Shared**: variables with matching names are shared between paired components of the same type, while variables can also be paired explicitly between arbitrary component types.
- **Local**: a local variable is never synchronized among any components.

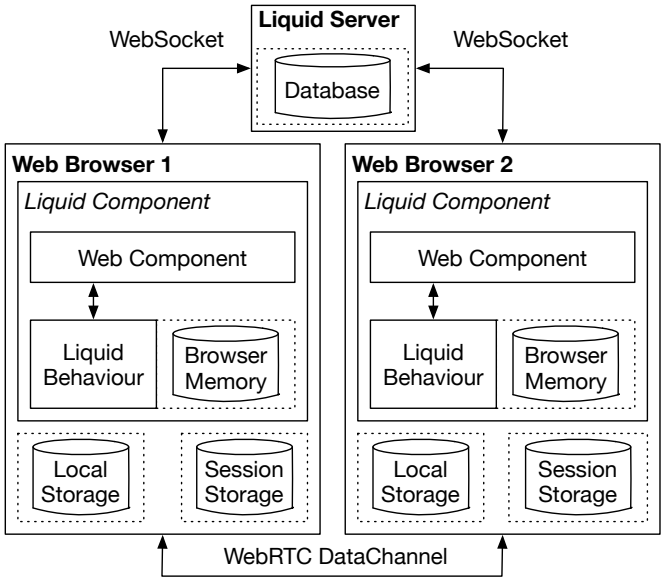


Fig. 3: Liquid.js runtime and storage mechanisms

2) **Component Scope**: The component scope defines whether variables are shared among instances of the same component type (**Intra-component**) or among instances of any type of Liquid components (**Inter-component**).

3) **Device deployment**: The device deployment defines if a variable can be shared across multiple devices.

- one device: a variable can be shared with components which are deployed and instantiated on the same Web browser.
- many devices: a variable can be shared with any component, also if it is running on another Web browser (on the same or different device).

4) **Persistence Policy**: The persistence defines how long a variable should be stored. We distinguish:

- **Persistent**: the value of a variable is permanently stored even if all instances of the Liquid components containing it are closed or all the devices are shut down.
- **Session**: the value of the variable is stored until the user closes every Web browser running one instance of the component containing the variable.
- **Volatile**: the value of the variable is stored until at least *one instance of a component* holding the value exists.

The composition of these four dimensions makes it possible to automatically decide where the value of a variable should be stored by the Liquid.js framework. This opens up the opportunity to minimize the bandwidth consumption and the latency of the synchronization.

We identified four possible places where the state of Liquid variables can be stored:

- **Browser Memory** (JS Heap): the Browser Memory is the best place to store volatile variables. Whenever the component is closed, the state of its volatile variables is lost.
- **Local Storage** allows to store and efficiently share the state among all components running on the same Web browser. If the local storage is not available in the Web browser, the server-side solution is used instead.

TABLE I: Storage mechanism chosen based on the sharing policy, component scope, device deployment and persistence policy of a Liquid Variable

	Scope	**Intra-component (same type)**		**Inter-component (different types)**	
	Deployment	**One Device**	**Many Devices**	**One Device**	**Many Devices**
Persistent	**Global**	Local Storage	Server-side	Local Storage	Server-side
	Shared	Local Storage	Server-side	Local Storage	Server-side
	Local	Local Storage			
Session	**Global**	Session Storage	Server-side	Session Storage	Server-side
	Shared	Session Storage	Server-side	Session Storage	Server-side
	Local	Session Storage			
Volatile	**Global**	Browser Memory	Browser Memory	Browser Memory	Browser Memory
	Shared	Browser Memory	Browser Memory	Browser Memory	Browser Memory
	Local	Browser Memory			
Persistence Policy	Sharing Policy				

- **Session Storage**: Unlike local storage, session storage discards the state when a session ends. If the session storage is not available in the browser, the server-side solution is used instead.
- **Server-side Storage**: Global variables potentially require all devices to access and synchronize their state. To do so, we implement server-side storage using the Liquid Server since this is directly connected to all instances of all Liquid components. The Liquid Server is the only component of the runtime which may survive disconnection of all client devices, and thus can safely store the state of persistent global variables. Global variables with a session persistence policy are also retrieved from the Liquid Server, but synchronized directly between devices. The server will discard their value once all sessions end on all devices.

Liquid variables are mapped to the corresponding storage mechanism based on the four dimensions discussed previously. Table I summaries where the state of a variable should be stored for each valid combination. The synchronization mechanism is implemented directly in the Liquid components where the value of a variable is cached. Liquid components directly exchange messages through WebRTC DataChannels in multi-device communications (if available) or by using internal messages during single-device synchronization. The different storage we proposed are used to make data persistent. Synchronization is carried out by the Liquid Server for global persistent variables, while shared variables are synchronized using the peer-to-peer mesh built with WebRTC among all devices. This way, it is possible to make data persistent while reducing the bandwidth consumption of the server.

Figure 3 summarizes all the possible places where Liquid.js stores Liquid variables to keep the data as close as possible to the component using it. The volatile data is stored in the browser heap memory inside the Liquid Component. Whenever the data does not need to be persistent, it can be stored inside the Web browser in both Local and Session storage (if available). Whenever the data is persistent it is stored in the database connected to the Liquid Server so that it can be accessed also from other devices.

IV. Discussion

With Liquid.js we provide the basic infrastructure for Web developers that would like to add support for the liquid user experience while taking advantage of emerging standard Web component technology. In this section we discuss issues that we gathered while assessing how to ease the adoption of Liquid.js in practice.

A. Security

Devices of the same user discover one another using standard pairing techniques [21]. Communication channels between devices and the Liquid Server can be secured using standard cryptographic techniques. Applications built using Liquid.js can precisely control which information is shared among multiple devices through the *publish-subscribe* variable permissions, (Section III-C) which require both parties to agree to establish a pairing.

For collaborative scenarios involving multiple users, it is possible to reuse a similar mechanism for allowing users to control which information is revealed as Liquid components cross user ownership boundaries. Likewise, a device may reject the dynamic deployment of unauthorized Liquid components flowing from other untrusted users.

B. Synchronization and Latency Issues

Liquid.js optimises the transport bindings among the Liquid components. The priority of the channels is the following: if possible Liquid.js will try to synchronize the data locally; if local communication channels are not enough it will try to share data through the peer-to-peer RTCDataChannels; if peer-to-peer is not available it will relay messages through a central Web-server. In the case of collaborative scenarios, conflict resolution is taken care by *Yjs* [19].

C. Liquid User Experience

The Liquid frame default implementation can be overridden. Not only the Liquid frame is optional, but the developer can completely redefine it by using the component deployment lifecycle API. In particular, we are using a push-based approach, where users interact with existing instances of a component to move them elsewhere. We are also experimenting with pull-based approaches, where the target device is used to choose the component instance that should be migrated on it.

D. Scalability

As the number of devices grows, a method to classify them is needed, not only according to their capabilities but also according to the role they are expected to perform in the application. This is particularly useful in parallel screening scenarios where multiple similar devices (e.g., smart phones) are supposed to present complementary views (the presenter broadcasting content vs. the viewers consuming it) or access channels to the same Web application. This would simplify capturing the deployment configuration of the Liquid components, which is currently specified targeting individual devices.

V. Conclusions and Future Work

Liquid.js for Polymer is a framework that simplifies the creation of component-based Web Applications featuring a liquid user experience. The framework provides developers with an explicit, declarative definition of Liquid components, encapsulating user interactions, behavior, and their state. These components can be deployed among standard Web browsers running on multiple devices through the provided Component Deployment Lifecycle API, which provides support for creation, migration, cloning and destruction of the components across all devices owned by one or more users. Stateful components are transparently synchronized whenever multiple instances are cloned and deployed on multiple paired devices. In this paper we presented the design of the framework focusing on its liquid state storage management features.

While some solutions [24], [17] to make components responsive and enhancing their adaptability already exists, we are trying to automatically add the adaptation to the heterogeneous devices in the framework, in such a way that Liquid.js will be able to take advantage of all the collective capabilities of all devices. Moreover, the ultimate goal is to liquefy all layers of an application and not only in its stateful user interface components. In the current design a component uses only the local resources of the device instantiating it, but in our vision we want to make paired devices share also the logic layer of the application. This would help to minimize redundant computations and avoid unnecessary conflict resolution over the shared state. To do so, we plan to investigate how to dynamically deploy component logic by finding the most suitable execution environment to make it possible for non-visual components of liquid Web applications to transparently share the aggregated computational power of all connected devices.

Acknowledgments

We are grateful for valuable feedback of George Fairbanks and the anonymous reviewers. This work is partially supported by the SNF and the Hasler Foundation with the Fundamentals of Parallel Programming for Platform-as-a-Service Clouds (SNF-200021_153560) and the Liquid Software Architecture (LiSA) grants.

References

[1] D. Bonetta and C. Pautasso. An architectural style for liquid web services. In *Proc. of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 232–241, 2011.

[2] D. Bourges-Waldegg, Y. Duponchel, M. Graf, and M. Moser. The fluid computing middleware: Bringing application fluidity to the mobile internet. In *IEEE/IPSJ International Symposium on Applications and the Internet (SAINT'05)*, pages 54–63, 2005.

[3] S. Casteleyn, I. Garrigós, and J.-N. Mazón. Ten years of rich internet applications: A systematic mapping study, and beyond. *ACM Transactions on the Web (TWEB)*, 8(3):18, 2014.

[4] O. Chudnovskyy, C. Fischer, M. Gaedke, and S. Pietschmann. Inter-widget communication by demonstration in user interface mashups. In *Web Engineering*, pages 502–505. Springer, 2013.

[5] O. Chudnovskyy, S. Pietschmann, M. Niederhausen, V. Chepegin, D. Griffiths, and M. Gaedke. Awareness and control for inter-widget communication: challenges and solutions. In *Web Engineering*, pages 114–122. Springer, 2013.

[6] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *Software Engineering, IEEE Transactions on*, 24(5):342–361, 1998.

[7] A. Gallidabino and C. Pautasso. The liquid.js framework for migrating and cloning stateful web components across multiple devices. In *Demo accepted at WWW'16*, 2016.

[8] A. Gallidabino, C. Pautasso, V. Ilvonen, T. Mikkonen, K. Systä, J.-P. Voutilainen, and A. Taivalsaari. On the architecture of liquid software: technology alternatives and design space. In *accepted at WICSA'16*, 2016.

[9] Google. The connected consumer. http://www.google.com.sg/publicdata/explore?ds=dg8d1eetcqsb1_, 2015.

[10] J. Grundy, X. Wang, and J. Hosking. Building multi-device, component-based, thin-client groupware: issues and experiences. In *Australian Computer Science Communications*, volume 24, pages 71–80, 2002.

[11] D. Guinard, V. Trifa, and E. Wilde. A resource oriented architecture for the web of things. In *Internet of Things (IOT), 2010*, pages 1–8. IEEE, 2010.

[12] J. Hartman, U. Manber, L. Peterson, and T. Proebsting. Liquid software: A new paradigm for networked systems. Technical report, Technical Report 96, 1996.

[13] J. J. Hartman, P. Bigot, P. Bridges, B. Montz, R. Piltz, O. Spatscheck, T. Proebsting, L. L. Peterson, A. Bavier, et al. Joust: A platform for liquid software. *Computer*, 32(4):50–56, 1999.

[14] B. Kemme and G. Alonso. Database replication: a tale of research across communities. *Proc. of the VLDB Endowment*, 3(1-2):5–12, 2010.

[15] G. E. Krasner, S. T. Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[16] M. Levin. *Designing Multi-device Experiences: An Ecosystem Approach to User Experiences Across Devices*. O'Reilly, 2014.

[17] E. Marcotte. *Responsive web design*. A Book Apart, 2011.

[18] T. Mikkonen, K. Systä, and C. Pautasso. Towards liquid web applications. In *Proc. of ICWE*, pages 134–143. Springer, 2015.

[19] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma. Yjs: A framework for near real-time p2p shared editing on arbitrary data types. In *Proc. of ICWE*, pages 675–678. Springer, 2015.

[20] J. Pérez, M. Murray, G. Coffin, J. Fluker, and Z. Bailes. Connectivity and continuity: New fronts in the platform wars. In *Panel at Twenty-First Americas Conference on Information Systems (AMCIS)*, 2015.

[21] N. Saxena and M. B. Uddin. Automated device pairing for asymmetric pairing scenarios. In *Information and Communications Security*, pages 311–327. Springer, 2008.

[22] K. Systä, T. Mikkonen, and L. Järvenpää. HTML5 agents: Mobile agents for the web. In *Web Information Systems and Technologies*, pages 53–67. Springer, 2014.

[23] A. Taivalsaari, T. Mikkonen, and K. Systa. Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. In *Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual*, pages 338–343. IEEE, 2014.

[24] J.-P. Voutilainen, J. Salonen, and T. Mikkonen. On the design of a responsive user interface for a multi-device web service. In *Proc. of the Second ACM International Conference on Mobile Software Engineering and Systems*, pages 60–63, 2015.