Liquid Web Applications

Design and Implementation of the Decentralized Cross-Device Web

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Andrea Gallidabino

under the supervision of
Prof. Cesare Pautasso

June 2020

Dissertation Committee

Prof. Maristella Matera	Politecnico di Milano, Italy
Prof. Tommi Mikkonen	University of Helsinki, Finland
Prof. Marc Langheinrich	Università della Svizzera italiana, Lugano, Switzerland
Prof. Michele Lanza	Università della Svizzera italiana, Lugano, Switzerland

Dissertation accepted on 25 June 2020

Research Advisor	PhD Program Director
Prof. Cesare Pautasso	**Prof. Dr. Walter Binder, Prof. Dr. Silvia Santini**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Andrea Gallidabino
Lugano, 25 June 2020

Learn this lesson, that to be self-contented is to be vile and ignorant, and to aspire is better than to be blindly and impotently happy.

Edwin A. Abbott

Abstract

Web applications are traditionally designed having in mind a server-centric architecture, whereby the whole persistent data, dynamic state and logic of the application are stored and running on a Web server. The clients running in the Web browsers traditionally render only pre-computed views fetched from the server.

Nowadays this centralized approach does not fit well with the kind of interactions that the users perform when they connect to a Web application. The users can access the Web and fetch applications with much faster devices than the ones we owned thirty years ago. Moreover the Web can now be accessed with devices of any shape, size, and capability: ranging from desktop computers, to laptops, tablets, and smartphones. Emerging smart and embedded devices in the Internet of Things are also able to access the Web and interact with each other thanks to new emerging Web standards, such as smart televisions, smart watches, or smart cars. The diversity in the devices increased together with the average number of Web-enabled devices owned by a single user. Today the average connected users access the Web with multiple devices at the same time and expect that their applications, which are now deployed on all the devices they own, can be seamlessly used on each one of them.

In this dissertation we discuss liquid Web applications: software that can be deployed in a cross-device environment by exploiting the current HTML5 standards. In particular we design and implement decentralized liquid Web software able to flow between different platforms. Like liquid adapts its shape to its container, liquid Web applications adapt and can be deployed on all available devices. The Web platform allows devices of different manufactures to communicate, deploy, and distribute liquid applications among them, even when they do not share a common operating system. With liquid Web applications we seek to overcome the current stagnation in the traditional design of *solid* Web applications in favor of an affordable cross-device solution.

We present the history and evolution of liquid applications and discuss why the Web is the best platform for creating them. We show how to design liquid

software by discussing how to deploy the state, logic, and User Interface (UI) of any Web application on multiple devices. The design we present allows developers to create liquid Web applications able to seamlessly flow between multiple devices following the attention of the users. We also present the Liquid.js for Polymer framework, whose goal is to simplify the creation of liquid Web applications by helping developers to create their own Liquid User Experience (LUE). Our contribution in the design of liquid software presented in this dissertation is decoupled from the framework implementation and can be re-used to create new liquid frameworks.

Acknowledgements

Many people have helped me finish this dissertation, to whom I would like to warmly express my sincere gratitude and recognition.

I cannot start this section without being grateful to my advisor Prof. Cesare Pautasso. He trusted my abilities and gave me the opportunity to work with him and finish this dissertation in the best environment I could have ever imagined. He taught me so many useful skills, some that go even beyond the scope of software design. For this reason I recognize in him a great mentor that goes beyond the educational system we live in. I will never thank you enough for helping me throughout these years, for helping me sorting my chaotic ideas, and produce the quality publications we worked on together. I am especially thankful for all the time you committed to helping me writing and present my work.

I would like to thank Prof. Marc Langheinrich and Prof. Michele Lanza from USI, Prof. Maristella Matera from Politecnico di Milano, and Prof. Tommi Mikkonen from University of Helsinki for their support. Their feedback helped me write this dissertation.

I met so many competent people during this journey, many of whom I met while travelling around the world. The list is long, and it would be impossible to thank all of them, but I still carry their advices with me. All your inputs helped me shape this dissertation.

A big thank-you to my colleagues Masiar Babazadeh, Vincenzo Ferme, Ana Ivanchikj, and Vasileios Triglianos for creating the best environment in our office. We helped each other and we had so much fun. Especially I would like to acknowledge the help of Masiar Babazadeh, who made me discover and love the beauty of distributed Web applications during my master thesis, a love that I still nurture in my heart today. Obviously I will not forget to mention the good time we spent together the past years: drinking coffee and slaying monsters.

I would also like to extend my acknowledgements to all the people I met at USI, students included. I learned so much from all of them, and they made me become a better person.

I dedicate this dissertation to my family and my beloved, without their sup-

port I would not be the person I am today.

Finally, my last acknowledgements go to Antonov, Gigietto, Rinik, Robb, and Sithar, the true heroes of my story.

Contents

Figures

Tables

Part I

Liquid Software

Chapter 1

Introduction

1.1 Motivation

The way users interact with their devices has radically changed in the last three decades, and this is especially true when we investigate the way users interact with the devices connected to the Web. Nowadays the number of Web-enabled devices is growing fast [IoT18], while the price for connecting them to the Web is in average cheaper and more affordable than it previously was [Wor14; Wor19]. Hardware is cheaper and the average user now owns more than three Web-enabled devices [Glo17].

With the access to more powerful and capable devices, the majority of the users access the Web with multiple devices simultaneously [Goo17a], using various type of appliances [Goo17b]: ranging from common desktop computers, tablets, smartphones, and Internet of Things (IoT) devices, to more recent smart televisions, cars, and cutting edge wearable devices [146].

Depending on the situation [101; 102] and on the devices in reach [43], the users access the Web differently and prefer to use some devices more than others to perform their activities [97]. E.g., when the users are in a office they interact more with multiple desktops and laptops, while when they are at home they usually interact more with tablets and smartphones.

We are currently living in the multiple device ownership era [174] [And15], in which software applications that traditionally were meant to merely run on a personal computer, now are expected to run on any kind of device, as it has been already discussed by Dearman *et al.* in 2008 [48]. The way we interact with the software has changed accordingly, and we recognize new usage patterns when we run multi-device applications [145; 150; 91]. Surrounded by multiple devices, the users assume they have the ability to access their applications from

any device they own [100], and they assume the data is automatically available on any device they decide to use, even when they decide to use them simultaneously [116].

However, these features are not automatically implemented in multi-device applications and the presence and usage of multiple devices add several degrees of complexity to the users that do not know how to use them, as well as to the developers in charge of designing them. Application versioning consistency management [152] and cross-device state synchronization [180; 5] are two examples of problems that arise from multi-device interactions.

From the users perspective, managing a set of devices as independent entities from within the same application is confusing, nevertheless breaking this boundary and transparently allow applications to run on a set of devices, gets more complex as the number of connected devices increases [12]. The users must be in control of the applications and they must be able to manage them on any device.

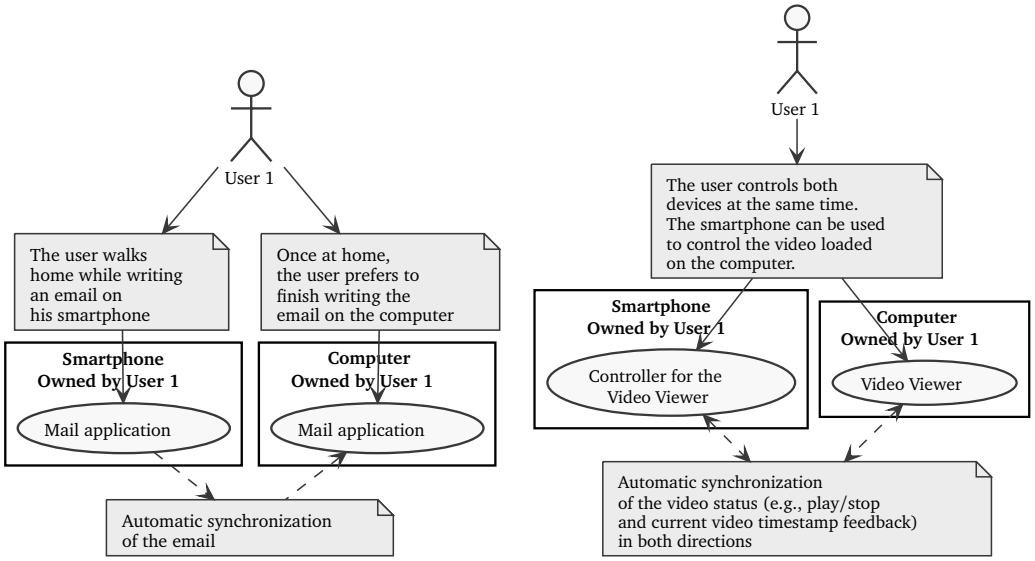
From the developer perspective, traditional applications are not designed for spanning the user experience on multiple devices [168]. Operating systems, virtual machines, applications and even components within applications are designed to be running on a single device and thus their user experience is designed for single device usages only [30; 85]. E.g., the users install applications on a device independently, and they manage the set of applications installed for each one of them independently.

We need to overcome the boundaries of single device file systems and share settings among devices if we want to truly create seamless multi-device user experiences, and it is clear that the cost for managing multi-device applications and ensuring that the devices can transparently run them simultaneously, grows as the number of connected devices increases.

For this reason we introduce the concept of liquid software, a paradigm used to explain how to design applications that may run seamlessly on a set of devices, and that are able to adapt their behavior in accordance to their deployment context. The Liquid User Experience (LUE) covers the following three scenarios [Goo12] (see Figure 1.1):

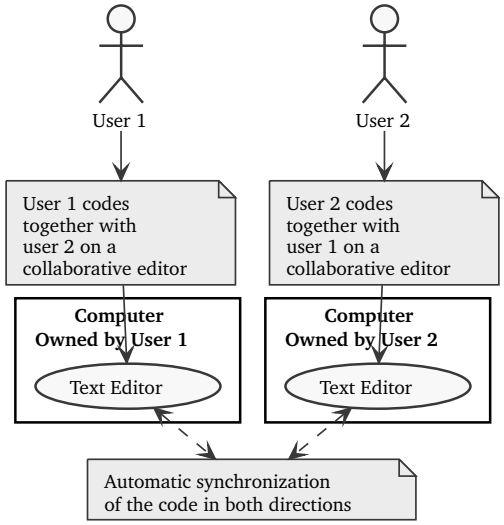
• **Sequential Use** (Figure 1.1a) - a single user runs an application on different devices at different times. The application adapts to the different devices capabilities while respecting the actual user's needs in different usage contexts;

• **Simultaneous Use** (Figure 1.1b) - a single user interacts with an application from several devices at the same time, e.g., the session is open and running on multiple devices at same time. Different devices may show an adapted view of the same User Interface (UI), or the system may have a distributed or cross-



(a) Sequential screening: writing an email.

(b) Simultaneous screening: watching videos.



(c) Multi-user, multi-device collaboration: collaborative coding.

Figure 1.1. Liquid software use-case scenarios.

device [88] UI in which different devices play their own distinct roles;

- **Collaborative Use** (Figure 1.1c) - several users run the same application on their devices and collaboratively work on the same data synchronized between them. This collaboration can be either sequential or simultaneous [90; 128].

The scenarios share the same technical challenges in adapting the UI to dif-

ferent devices and in synchronizing the data and state of the execution between devices. The synchronization of the data and state across multiple-devices is fundamental in the implementation of liquid software [132], because the devices and users need to be aware of the results of their actions previously or simultaneously done on other devices. This is essential for transferring the work from one device to another, thus enabling seamless, real-time device usage.

A truly liquid software ecosystem should support heterogeneous devices across native software ecosystem boundaries (e.g., across devices running different Operating Systems (OSs)). This way, developers would be able to implement an application only once, which could then adapt itself to run on various types of devices [57]. A viable option in realizing such a vision is to leverage the Web ecosystem, where applications are already deployed on demand and can adapt and fit to the devices' specifications with responsive Web design [125]. Properties such as openness and freedom from proprietary features make the Web a natural choice over native applications that are bound to a particular operating system, manufacturer, or vendor-specific ecosystem (e.g., vendor-specific smartphones) [134].

1.2 Research Questions (RQs)

The contributions brought by this dissertation to the field of cross-device Web applications design seek to answer the five Research Questions (RQs) described in Table 1.1.

1.2.1 RQ#1 - Liquid Software Design

How can we help Web developers design liquid software and the LUE?

Liquid Web applications are complex applications able to interact with multiple devices simultaneously. The complexity of these interactions is transparently hidden behind the LUE and requires real-time message passing and data synchronization across all the connected devices in order to seamlessly move applications among them. Liquid Web software can be designed following different quality drivers and the design can be reflected into a multitude of different software architectures depending on the design alternatives that are selected during the design phase. What are the design alternatives related to liquid Web applications? What are the implications of such choices?

In this dissertation we list and discuss the design alternatives related to liquid software and present past, present and envisioned technologies that can be used

Table 1.1. List of RQs.

#	**Research Question**	**Chapter(s)**
1	How can we help Web developers design liquid software and the Liquid User Experience (LUE)?	Section 1.2.1 Section 3.2 Chapters 4, 5, 6
2	How can we abstract liquid Web applications away from the current strongly centralized deployment approaches?	Section 1.2.2 Section 3.3 Chapters 4, 5, 6
3	How can we make the Distributed User Interfaces (DUI) of a Web application automatically adapt to the set of connected devices?	Section 1.2.3 Chapters 4, 5, 6
4	How can we take advantage of all resources provided by the set of connected devices?	Section 1.2.4 Chapters 7, 8, 9
5	How can we design secure liquid Web applications? How can we enhance privacy?	Section 1.2.5 Section 7.6

for developing it. Quality attributes and design decisions are discussed by presenting both the advantages, drawbacks, and impact in the overall architecture of the liquid application. We validate the proposed design decisions by applying them in the implementation of our liquid software framework.

1.2.2 RQ#2 - Beyond Centralized Deployments

How can we abstract liquid Web applications away from the current strongly centralized deployment approaches?

The vast majority of existing Web applications implementing the LUE relies on centralized architectures and deployments, however interactions of the users happening on their own set of devices do not necessarily need to pass through a Web server, in fact devices can communicate directly with each another with Peer-to-Peer (P2P) channels. From the users' perspective, they can preserve privacy when messages are not sent to a Web server, since the users' data and interactions can be modelled to be synchronized independently from a centralized Web server of Cloud Service outside of the user's control. What are the trade-offs of this approach?

In this dissertation we design liquid Web software with a decentralized architecture, shifting away from the more common centralized one. When possible,

given the limitations of the current Web technology standards, the design decisions are taken considering a distributed environment. The proposed prototype reflects the decentralized nature of these design decisions.

1.2.3 RQ#3 - LUE Adaptation Among Devices

How can we make the DUI of a Web application automatically adapt to the set of connected devices?

Developers of liquid software should provide to the users a mechanism for populating the set of devices with parts of the liquid applications. For this reasons we must provide to the developers the tools for developing automatic complementary views. In Web applications meant to have complementary views, the UI can scatter and adapt across multiple devices (e.g., each device displays different parts of the application). This allows the developers to have a certain degree of control on how the application is migrated across the devices, instead of exclusively allow the users decide how the deployment of the application will evolve over time. A misuse of the manual LUE may lead to non-intuitive deployments which contradict the developers' expectations and intent.

In this dissertation we discuss how to deploy and adapt liquid software across multiple devices by separating its internal representation into three interconnected layers: • `data layer` - which defines the data and state of a liquid applciation; • `logic layer` - which defines the controlles of a liquid application; • `view layer` - which defines the UI of a liquid application. For each individual layer we present how it can be manually deployed across multiple Web-enabled devices and then discuss how their deployment can be automated following the design decisions of the developers, or the needs of the users when applicable.

1.2.4 RQ#4 - Sharing Resources Among Devices

How can we take advantage of all resources provided by the set of connected devices?

Any Web-enabled device has access to at least a screen, a data storage and a Central Processing Unit (CPU) (in the case of small IoT devices, it is possible that they lack a screen and have access to a limited data storage and CPUs). While the LUE takes advantage of all the screens and the decentralized synchronization can take advantage of all the provided data storage, we need to further develop a mechanism for efficiently exploit the CPUs of all connected devices in the liquid software environment. Without such a mechanism all the devices compute

the same operation multiple times, even if it would be better to compute the operation only once on a single device and then broadcast the result.

In this dissertation we show how executable tasks can be shared across the set of Web-enabled devices and how their result can be broadcasted back to all participants in a liquid Web application.

1.2.5 RQ#5 - Privacy and Security

How can we design secure liquid Web applications? How can we enhance privacy?

As liquid Web applications can dynamically flow between multiple devices owned by different users, it becomes important to consider the security and privacy implications of liquid software. In this dissertation we design decentralized liquid software, where the data and the computations generated by the liquid applications are moved from the Web servers towards the clients. As we shift from traditional server-client to peer-to-peer architectures, we need to design secure communications between the connected peers and ensure that the users have control on the data stored in their Web browsers. For this reason we need to design a distributed access control approach whose goal is to avoid that both the data and the runtime state of rich component-based Web applications is stored in devices not fully owned by their users and to prevent malicious users from accessing them.

In this dissertation we design a secure and privacy-preserving architecture for component-based liquid Web applications, and discuss how we build it on top of our prototype. The design specifies a capability-based Discretionary Access Control (DAC) model in which users can manage the flow of data migration between devices owned by multiple users in collaborative scenarios.

1.3 Summary and Outline

This dissertation is divided in three parts and ten chapters:

• **Part I - Liquid Software**. In the first part of this dissertation we introduce the liquid software paradigm and the reasons it is relevant today. The discussion is divided into three chapters:

– **Chapter 1 - Introduction**. In the first chapter we present the technological impetus that influenced the definition of the liquid software paradigm. We present the challenges derived from the paradigm and discuss what today's liquid applications are lacking. We discuss the benefits that can be derived by evolving

the liquid software paradigm and then define the motivations behind this dissertation. This chapter also presents the RQs answered throughout the following chapters, the related publications and contributions of this work.
– Chapter 2 - State of the Art. In this chapter we present the liquid software paradigm, the meaning of the *liquid* metaphor, and then present other uses of the metaphor in other fields. We present the history of liquid software, starting from the inventors of the term and follow up the discussion with all the works derived from their initial vision. We also explore the most related research areas strongly bound to the liquid software paradigm and present their work. Finally we present some existing liquid software and technologies and their impact in the industry.
– Chapter 3 - Liquid Software Design. In this chapter we present the design considerations and alternatives related to liquid software. We focus the discussion on the Web platform and derive the maturity model of the current existing liquid Web technologies. We also list the quality attributes, challenges, advantages and drawbacks of multiple design decisions and their impact on the liquid software design.

- **Part II - Liquid Web Architectures**. The second part of this dissertation focuses on the design of liquid Web applications. We abstract and divide liquid software into three interconnected layers and discuss how we can design them in a liquid application. Each chapter discusses one of the three layers:

– Chapter 4 - Liquid Data Layer and State Synchronization. The data layer holds both the data and state of an application. In this chapter we discuss how we can deploy and automatically synchronize the data of liquid Web applications across multiple devices. We present the set of technologies that can be used in the design of the liquid data layer and discuss about multiple deployments of the liquid storage.
– Chapter 5 - Liquid Logic Layer and Liquid WebWorkers. In this chapter we discuss how the set of devices connected to a liquid application can offload executable tasks among each other. By exchanging tasks, we allow the logic layer to be deployed on any device participating in the liquid application. We design the liquid logic layer by presenting the scenarios, challenges, technologies and the possible design decisions that must be considered for deploying this layer across multiple devices.
– Chapter 6 - Liquid View Layer and Liquid Media Queries. In this chapter we discuss how we can automatically deploy UIs across multiple devices. We present the current technologies used for creating Web UIs and the technological gap that need to be filled in order to create a view layer that can span across multiple devices. We discuss how we plan to update the current standards and

propose rules and algorithms that can automatically deploy the UI of a liquid application on a set of devices.

• **Part III - Framework Implementation**. The third part of this dissertation presents our implementation of a liquid software framework following the design presented in the second part. This part is divided into three chapters:

– Chapter 7 - Liquid.js for Polymer. In this chapter we present our framework that can be used to develop liquid Web applications that have transparent access to the LUE. We present the framework abstractions and for each layer we discuss its implementation. Furthermore we discuss privacy and security implications derived by the framework.

– Chapter 8 - The Liquid.js Application Programming Interface (API). In this chapter we present the Liquid.js API that developers can use to develop their own liquid applications.

– Chapter 9 - Validation. In this chapter we evaluate the performance of the liquid data layer and logic layer of our implementation. We then validate the expressiveness of the liquid view layer and discuss how developers can create their own Web application featuring the LUE primitives.

– Chapter 10 - Conclusions. The final chapter of this dissertation summarize the answers to the RQs proposed in the first chapter. We finally discuss the limitations of our framework and the possible future works that can follow this dissertation.

1.4 Contributions and Publication Overview

The main contribution of this dissertation is the implementation of Liquid.js for Polymer, our prototype framework that encapsulates all design decisions discussed throughout this dissertation. The design aspects and the actual implementation of the framework can be separated from one another and thus the design of the liquid layers of an applications are discussed independently from the framework. This dissertation gives to the cross-device research area the following contributions:

• The definition of the design space of liquid applications [60; 61];

• The definition of the maturity model of liquid software [68];

• The implementation of Liquid.js for Polymer and the exposed API that developers can exploit for creating their own liquid applications and the LUE [70];

• The design of the storage deployment for liquid Web applications [66];

• The design of liquid media query that extended Cascading Style Sheets 3 (CSS3) media queries [72; 73];

Table 1.2. List of publications related to this dissertation presented on conferences or published on journals.

	Publications
2016	**Andrea Gallidabino** and Cesare Pautasso. Deploying Stateful Web Components on Multiple Devices with Liquid.js for Polymer. *Proceedings of the International SIGSOFT Symposium on Component-Based Software Engineering (CBSE16)*, pages 85-90, 2016, IEEE. [66] **Andrea Gallidabino**, Cesare Pautasso, Ville Ilvonen, Tommi Mikkonen, Kari Systä, Jari-Pekka Voutilainen and Antero Taivalsaari. On the Architecture of Liquid Software: Technology Alternatives and Design Space. *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA16)*, pages 122-127, 2016, IEEE. [60]
2017	**Andrea Gallidabino** and Cesare Pautasso. Maturity Model for Liquid Web Architectures. *Proceedings of the International Conference on Web Engineering (ICWE17)*, pages 206-224, 2017, Springer. [68] **Andrea Gallidabino**, Cesare Pautasso, Ville Ilvonen, Tommi Mikkonen, Kari Systä, Jari-Pekka Voutilainen and Antero Taivalsaari. Architecting Liquid Software. In: *Journal of Web Engineering (JWE)*, Volume 16.5&6 pages 433-470, 2017, River Publishers. [61]
2018	**Andrea Gallidabino** and Cesare Pautasso. The Liquid User Experience API. *Proceedings of the The Web Conference 2018 Companion (TheWebConf2018 a.k.a. WWW18)*, pages 767-774, 2018. [70] **Andrea Gallidabino** and Cesare Pautasso. Decentralized Computation Offloading on the Edge with Liquid WebWorkers. *Proceedings of the International Conference On Web Engineering (ICWE18)*, pages 145-161, 2018, Springer. [69] **Andrea Gallidabino** and Cesare Pautasso. The Liquid WebWorker API for Horizontal Offloading of Stateless Computations. In: *Journal of Web Engineering (JWE)*, Volume 17.6 pages 405-448, 2018, River Publishers. [71]
2019	**Andrea Gallidabino** and Cesare Pautasso. Multi-Device Adaptation with Liquid Media Queries. *Proceedings of the International Conference On Web Engineering (ICWE19)*, pages 474-489, 2019, Springer. [72]
2020	**Andrea Gallidabino** and Cesare Pautasso. Multi-Device Complementary View Adaptation with Liquid Media Queries. In: *Journal of Web Engineering (JWE)*, Volume 18.8 pages 1-40, 2020, River Publishers. [73]

Table 1.3. List of additional related publications.

	Publications
2016	**Andrea Gallidabino** and Cesare Pautasso. The Liquid.js Framework for Migrating and Cloning Stateful Web Components across Multiple Devices. *Proceedings of the International Conference on the World Wide We Companion (WWW16)*, pages 555-558, 2016. Demonstration [67]
	Andrea Gallidabino. Migrating and Pairing Recursive Stateful Components Between Multiple Devices with Liquid.js for Polymer. *Proceedings of the International Conference on Web Engineering (ICWE16)*, pages 555-558, 2016, Springer. Demonstration [63]
2017	**Andrea Gallidabino**, Tommi Mikkonen, Niko Mäkitalo, Cesare Pautasso, Kari Systä, and Antero Taivalsaari and Jari-Pekka Voutilainen. Liquid Web Applications: ICWE2017 Tutorial. *Proceedings of the International Conference on Web Engineering (ICWE17)*, pages 269-271, 2017, Springer. Tutorial [62]
2019	**Andrea Gallidabino**. Liquid Web Architectures. *Proceedings of the International Conference on Web Engineering (ICWE19)*, pages 560-565, 2019, Springer. Doctoral Symposium [64]

- The design of Liquid WebWorker (LWW) that wraps the Hypertext Markup Language 5 (HTML5) WebWorker standard [69; 71];
- The Discretionary Access Control (DAC) model for private and secure liquid Web applications.

Parts of this dissertation are derived from published peer-reviewed conference papers and journal articles. In Table 1.2 we list all publications published on journals and conferences. In Table 1.3 we list the demonstrations, tutorial, and doctoral symposium related to this dissertation.

Chapter 2

State of the Art

Liquid software cannot be summarized into a single specific technology, but it can be described as a set of features (that can be mapped into multiple technologies) that need to be designed into applications that are executed on multiple, heterogeneous computing devices [174]. Nowadays, the features we expect to be implemented in liquid software, e.g., automatic transparent synchronization on multiple devices, are not always supported by the majority of the existing applications, and when an ecosystems provide said features, the users have to explicitly turn them on.

Due to the decreasing in the price of the hardware, computers, smart-devices, and displays are more affordable and as a consequence today's life where a huge amount of devices surround us [187]. The way we interact with these devices has evolved in the past thirty years as we discover new ways of interacting with them (e.g., with ubiquitous smart-devices [150] or pervasive public displays [24]). The kind of interactions also depend on the environment in which they are installed: e.g., users interact with laptops differently if they are in private environments where only one or very few users interact with it (e.g., smart homes [10]), or if they are in public locations with a huge amount of people [58], or if they are in shared spaces where a group of selected people need to interact with each other in order to achieve a common goal [155] (e.g., in offices [162]).

Weiser *et al.* predicted in 1991 [184], in the same period when laptops started to be more affordable to the masses, that in our century computers would be ubiquitous [185] and that their integration with our everyday lives would be so seamless to be *invisible* to the human perception. Even if we can say that their prediction was correct, we still have to achieve the *invisible computing* environment as they described in their work. Quoting Weiser *et al.*:

> [184] "*Technologies weave themselves into the fabric of everyday life*

until they are indistinguishable from it."

We believe that in order to achieve this goal at its full potential, we need to increase our efforts and invest more time in studying the development of cross-device applications [137] that aim to be so seamlessly integrated with the surrounding world. We believe that we did not achieve the goal at is fullest potential, because we spend more time designing distributed UIs [49], while we forget to develop a solid backbone application that can support liquid features. The whole application, as well as the UI, must be able to seamlessly flow in the invisible computing environment, so that we can create smart environments (e.g., EasyLiving from Microsoft [166]).

If we follow the history of computer evolution from 1991, the relationship between users and their computing devices has evolved from multiple users sharing one expensive and large computer to the far opposite, in which multiple, cheap, mobile and Web-connected devices are owned by a single user. Web browsers are nowadays ubiquitous as they run on desktop computers, laptops, tablets, smart phones, digital cameras, smart televisions, cars, and, with some limitations, even kitchen refrigerators, watches and glasses.

The Web is the biggest and most accessible platform available today, as it allows developers to share and distribute information and applications with any device produced by any manufacturer. The newer and emerging HTML5 standards removed or soon will remove the remaining barriers that will allow developers to build any kind of application on the Web. Developers are steadily creating more and more Web-based software instead of focusing solely on the creation of binary platform-locked software [170]. The Web is not anymore just a place for storing information, but it also a full-fledged platform for creating software. No other platform can achieve the same level of world-wide distribution and portability. If a device can run a modern Web browser complying with the HTML5 standard, then it can run any application developed for the Web [9].

Nowadays, while responsive Web applications are designed to adapt to the different screen sizes and input/output capabilities of different devices, it is still challenging to develop rich Web applications which can seamlessly migrate across different user devices. For example, planning a trip on a large display and following the directions for the trip guided by the phone Global Positioning System (GPS); typing a short email by tapping on the phone screen but then as the email text grows longer deciding to continue the work on a computer with a real keyboard. As users begin to use multiple devices concurrently – for example: to watch television while looking up information on their tablet, to play games across multiple telephones, to share pictures taken with personal devices

and view them on a public display, to remotely control presentation slides from a watch, or to confirm a credit card transaction entered on a desktop computer using the fingerprint reader of the phone – only few Web applications fully take advantage of all available devices and distribute their UI accordingly or allow users to re-arrange different UI components at will.

Web developers and designers have successfully addressed the scalability challenge to serve Web applications to millions of users [4] and to personalize such applications to each user profile [20] (e.g., language, age, geographical location, regulatory constraints, etc.) and adapt it to the capabilities of their Web browsing device [125]. However, this has been under the assumption that users connect to use the Web application using one device at a time. Stateful Web applications which use cookies to establish a session with a particular Web browser make it difficult for users to switch devices in the middle of a browsing session [75]. Additionally, they may break when opening multiple tabs to run them and sometime assume that users logging in from different devices at the same time may indicate a security issue. In this dissertation we tackle these issues and study liquid Web architectures in order to create software able to adapt to a set of devices.

2.1 Liquid Software Metaphor

The term *Liquid Software* is derived from a metaphor: like a liquid adapts to the shape of its container, liquid software adapts to take full advantage of every device it is deployed on [174]. A liquid Web application [132] is able to flow between multiple devices following the attention focus of its users [116].

Hartman *et al.* [84] were the first to use the liquid metaphor to describe their new paradigm on active network in 1996. They envisioned that the source code of liquid software should be independent from the operating system platform and that applications must be able to support code that can be deployed on multiple machines, e.g., code snippets are allowed to be seamlessly transferred between devices. Location-independent code is the fundamental principle of liquid software, which is strongly related to the concept of *mobile code* [55].

Hartman *et al.* presented Joust in 1999 [85], a Java platform that allows mobile code [55] to be more maintainable, easier to update and to inject in the devices connected to the system. The metaphor has evolved over the years into the liquid software manifesto [174] presented in 2014 to emphasize specific properties of the user experience, instead of focusing only on how mobile code can be moved between devices in a active network [40]. Even if the metaphor

has evolved, it still focuses on the metaphorical **adaptability** property of a liquid at its core. The liquid software manifesto takes into consideration the fact that nowadays end users own multiple devices and that they wish to use all of them together to run their applications.

Liquid software can: 1. adapt the UI to the *set of* devices being concurrently used to run the application; 2. seamlessly migrate a running application across devices; 3. synchronize the state of the application distributed across two or more devices, effectively breaking down the continuity boundaries that exist between devices in proximity both in physical space as well as in cyberspace [174].

According to the manifesto of Taivalsaari *et al.*, software can be defined as *liquid* if it fulfils the following six core requirements:

1. "*In a truly liquid multi-device computing environment, the users shall be able to effortlessly roam between all the computing devices that they have.*" [174]

This requirement is fundamental for the liquid software paradigm: at any moment users can choose to change and operate any of their devices, in turn the liquid software must be able to roam and deploy itself on the new machine without making the users aware of all necessary underlying operations. The liquid users do not care how the liquid software is implemented and should be able to have access to the liquid features with as few configuration steps as possible.

2. "*Roaming between multiple devices shall be as casual, fluid and hassle-free as possible; all the aspects related to device maintenance and device management shall be minimized or hidden from the users.*" [174]

In theory, in the final evolution of the liquid software, the migration of an application is so seamless that when the gaze of the users point to a different device, the software is able to transparently roam from a device to another undetected. However this hypothetical scenario can easily break the privacy of the users, which would prefer to have some kind of control on the liquid software roaming. The users must be in control of the liquid software and must be able to configure it as they please, however once it is configured, the users do not need to know how it works.

3. "*The user's applications and data shall be synchronized transparently between all the computing devices that the user has, insofar as the application and data make sense for each device.*" [174]

Roaming an application between multiple devices implies that the data and the state of the application is copied and sent on multiple locations. The liquid software should move between devices only the needed state required to operate the application on another device. E.g., if the users wish to buy something online on their smartphone, the device needs to access the credit card information of the users, but does not need to access all the pictures stored on their desk-

top computer. As the user's data roams between devices, liquid software needs to synchronize the smallest quantity of data as possible, both for enhancing the overall performance of the application and for enhancing the privacy of the user.

4. "*Whenever applicable, roaming between multiple devices shall include the transportation / synchronization of the full state of each application, so that the users can seamlessly continue their previous activities on any device.*" [174]

The synchronization process should be able to handle any payload size, ranging from small data updates (Bytes) to the whole application state (MegaBytes). The LUE is enhanced by responsive and real-time application, as the users can keep working on their application without delay even if they suddenly change device. Optimizing the real-time synchronization of big chunks of data is a real challenge, which we cannot overcome when the upload bandwidth of a device is low or limited. Liquid software must design new kind of interactions that allow the users to operate the application even when some data is missing, while giving them the perception that the whole state is already migrated to the device.

5. "*Roaming between multiple devices shall not be limited to devices from a single vendor ecosystem only ideally, any device from any vendor should be able to run liquid software, assuming the device has a large enough screen, suitable input mechanisms, and adequate computing power, connectivity mechanisms and storage capacity.*" [174]

Liquid software must be free to roam to any device. Manufacturers and vendors always try to create their own locked ecosystems (e.g., Apple and Microsoft), but a truly liquid environment has no bounds. For this reason we aim to build liquid software on the Web, which can be accessed by the majority of all existing devices.

6. "*Finally, the user should remain in control of the liquidity of applications and data. If the user wishes certain functionality or data to be accessible only on a single device, the user should be able to define this in an intuitive, straightforward fashion.*" [174]

The users of liquid applications can decide which data is transferred to the devices they do not trust and can prevent at any time them roam of the data to certain devices. The users are in control of the data they produce.

In this dissertation we design liquid software which addresses all the six requirements discussed in the liquid software manifesto and as a first step towards achieving the liquid software vision, we also present the implementation of the Liquid.js for Polymer framework.

2.1.1 Similar Metaphors

The liquid metaphor is not only used to address liquid software, but is also used in different context. In 2011 Bonetta *et al.* use the liquid metaphor for defining liquid Web services [23]. In their work they discuss liquid architectural styles for creating Web services that can transparently scale when new resources are required and can be executed on multiple heterogeneous platforms. The main quality attributes of liquid Web services are deployability, composability, and scalability. Their work on Web services has a similar approach as the one we discuss in this dissertation, in fact the authors discuss fine-grained component-based architectural styles which we believe is also a solid design decision for creating liquid Web applications.

The liquid metaphor is also used by Babazadeh *et al.* in 2015 when they coined the concept of Web liquid streams [18]. The framework they propose can be used to deploy stream processing pipelines on volatile environments composed by multiple heterogeneous devices and IoT sensors. The framework they implemented orchestrates the creation and evolution of data streams that can dynamically change at any time during the execution. The framework is also in charge of creating Web Real-Time Communication (WebRTC) DataChannel [Moz20a] and redistributing resources between the set of connected devices even when it changes due to connections, disconnections, or failures. Upon failure the framework decides how to re-deploy the streams on the new environment, without losing data in the process. Their concept of Web liquid streams is similar to the liquid software paradigm because they both deal with deployments on volatile environment composed by multiple heterogeneous devices. However Web liquid streams focus more on the resource distribution for the stream processing and does not really give a solution for deploying the state and view of the application in a distributed environment.

A similar metaphor is used in *Fluid computing* [22], which denotes the replication and real-time synchronization of application states on several devices. The application state flows like a *fluid* between devices, similarly as we propose liquid software does. The authors list three main application areas where fluid computing is relevant: 1) multi-device applications, where several devices may be temporarily coupled to behave as one single device (e.g., a mobile and a stationary device); 2) mitigation of the effects of unreliable connectivity, where applications on ubiquitous devices can exploit full or intermittent connectivity; 3) collaboration, where multi-user applications enable several users to collaborate on a shared document. Technically the platform associated with fluid computing consists of middleware that replicates data on multiple devices, and achieves co-

ordination of these devices through synchronization. Each device has a replica of the application state, allowing the device to operate autonomously; a special synchronization protocol is used for keeping the replicas consistent. This approach is different from the liquid software paradigm which aims to minimize the synchronized state of the application in order to enhance the privacy of the users.

An other notable use of the liquid metaphor can be seen in liquid templates [Sho20] created by Shopify, which are used by many modern Web applications and frameworks. Liquid is a template engine meant to be easy and flexible to use which can create Web pages meant to run on any Web browser. The engine is non-evaling (it does not call the `eval` method) and templates can be edited by the users. In order to guarantee security, those templates are not evaluated on the Web server. While their goal is flexibility and security, the templating engine they propose is not close to the liquid software requirements we discuss in this dissertation.

On the opposite side of the metaphorical spectrum we have Solid [124], a project lead by Berners-Lee, the inventor of the World Wide Web. The name solid is derived from *social linked data*, and can be used to create decentralized social applications based on Linked Data. This project's goals is to further decentralize the Web, a topic that Berners-Lee [16] is discussing in many of his talks. For Berners-Lee the decentralized Web must be accessible by anyone from anywhere without stealing data from the users. In his vision the users have *true ownership* of the data they create and have freedom to choose where it is stored and have full control on who can access it. Solid decouples the content from the Web application, so that users can have control on the content even if they do not own the application. Even if the metaphor is not the same, we believe that the decentralization of the Web is important and required in the liquid software environment. Data privacy, portability, and deployability of the liquid applications are also at the core of the liquid software paradigm.

2.2 Beyond the Liquid Metaphor

While the term liquid software exists since 1996, it is not the only research area that discusses cross-device application deployments. Brudy *et al.* [29] surveyed more than 500 papers over 30 years of cross-device publications and noticed a lack of unified taxonomy in the multi-device research. For this reason cross-device deployments and cross-device interactions branched out in many different areas that use different terms for discussing the same challenges even if we aim at the same goal. This dissertation was primarily inspired by the liquid software paradigm, but it relates to many other concepts. Liquid software is not a single technology, rather a collection of features. In this section we discuss the main areas and related work to this dissertation.

2.3 Computer-Supported Collaborative Work (CSCW)

From the UI perspective, the roots of liquid software can be traced back to Computer-Supported Collaborative Work (CSCW) [80; 149], which focuses on enabling collaboration between multiple users with the help of one or more computing devices.

One of the many vision CSCW proposed in the past was presented by Stefik *et al.* in 1989 [161], in which they envisioned multiple users working in a office together with multiple devices. They discuss multiple scenarios in which users interact with each other, but instead of using pens and papers they own and share information through the devices they are using. These use-case scenarios are at the core of the liquid software paradigm.

Grundy *et al.* [83] presented in 2002 a collaborative, multi-device, component-based, thin client groupware system entirely based on Web technologies. The component-based allowed multiple users to perform activities together both on Web browsers and on mobile devices. Unfortunately with the technologies available in their era they could only achieve asynchronous interactions between the users because of technological limitations.

2.3.1 Cross-Device Interfaces

The cross-device distributed UIs research area discusses similar challenges to the CSCW, however it does not focus only on the multi-user use-case scenarios, but rather discusses the distribution of the view layer of applications in general.

XD-MVC [89] is a Web framework which can be used to develop decentralized cross-device applications focused on automatic cross-device adaptation of the application UI. The framework allows to easily decompose and migrate component-based Web applications built with the Polymer framework. The migration is implemented at the application level, but only the state is synchronized between devices. From the developer point of view, migration is implemented by clipping off child components from their parents, depending on which devices they are deployed, simulating the roaming behavior expected by liquid software. XD-MVC supports declarative adaptation of the view layer, as views and components can be annotated with rules that describe how components are expected to be shown across multiple devices. By interpreting up these rules, XD-MVC is able to decide which parts of a view need to be clipped off depending on the configuration of the set of connected devices. Husmann *et al.* [93] also implement a library that can be used to test if the UI of a cross-device application is visualized correctly on a distributed environment.

Nebeling *et al.* [137] proposes new techniques for enhancing cross-device interactions by creating a Graphical User Interface (GUI) builder designed to support interactive development of cross-device Web interfaces. The tool simulates a multi-device environment where the developers can design the application with an immediate feedback on what they are building.

Cross-device interfaces also deal with new ways for interacting with multiple devices. Di Geronimo *et al.* introduce Ctat [44], a new paradigms for interacting and sharing data between multiple mobile devices with intuitive gestures. With Ctat the Web browser of a smartphone device can detect the *tilt* and *tap* gestures performed by the users. In a cross-device environment the users can now tilt the phone towards an other device and tap it on the corner in order to perform a predefined action in the Web application.

Zorrilla *et al.* [194] propose to assign properties both to application components and devices. The centralized server uses these properties to score the best targets for the distribution, and then shows and hides the corresponding components depending on which devices they are deployed on. Similarly to their work, in this dissertation we use a rule-based approach for distributing Web application across multiple devices.

In the distributed UIs field, Santosa *et al.* [169] made a field study on the impact in the real world of the use of technologies enabling cross-device interactions. Given the responses of experts in the field, they collect and compare nine existing cloud-based data management software enabling cross-device collaboration between users.

Differently from the approaches proposed in the cross-device UIs area, in our

vision, design, and implementation of liquid software our goal is to be as decentrilzed as possible, without offloading any computation to the Web server. We also design liquid software that deploys as few components as possible on each of the available devices instead of deploying multiple copies of the application where they are not required. We also avoid to use cloud-based approaches so that the users can always be on control of their data.

In the literature we can also find several research topics that are similar to the cross-device UIs area, but with a different label, such as adaptive multi-device UIs [151] and DUI [115]. The majority of these researches focuses on native vendor-locked implementations or on centralized approaches.

2.3.2 Human-Computer Interactions (HCI)

The wider Human-Computer Interactions (HCI) area, tightly connected to the CSCW, describes a multitude of use-case scenarios for liquid software. Bellotti *et. al* [12] shows in 1996 how people interact and collaborate with each other in conjunction of their devices in a working environment. The survey proposed by Elmqvist [49] discusses the state of the art of distributed UIs in the HCI area. Elmqvist summarizes how it is possible to achieve migration of the UI and redirection of inputs and outputs of a multi-device deployment. The concept of redirection used by the authors is similar to the concept of forwarding inputs and outputs in liquid software.

HCI also defines many interactions that are useful to the liquid software paradigm. Ghiani *et al.* [76] distinguish how application can move from a device to another by separating them in two categories: *pull* and *push*. An application can be pulled by users from a device in their reach into their own device, while a push interaction means that the users can send their applications to a target device. These two operations have different implications in term of privacy and security. When an application is pulled by the users, they are requesting an application deployed on another machine and thus they do not care about the ownership of the state it stores, because they are not the owners. However the users must be aware of the security of the operation, since when the pull an application they must be aware that the software can be malicious. On the other hand when users push an application they care about privacy because the application they are pushing can contain data they created. Pull and push operations are an important in the design of liquid software.

Esenther [50] builds in 2002 a framework for real-time collaborative co-browser applications in the Web. The framework deploys a controller on the Web server of the application and allows two-way communication for synchronizing

the data between the devices in real-time.

2.3.3 Internet of Things (IoT) and Public Displays

Nowadays on average we own more than three Web-enabled devices per person [Glo17], this number is expected to further increase in the next years, foreseeing to reach the threshold of more than six devices per person in 2020 [Eva11]. Most existing Web applications are meant to run and be responsive to the specifications of a single device. With the increase of the average number of devices per user, there is a need to create applications able to run across multiple devices instead of just a single one at a time. Fog computing [21; 45] on the Web of Things [82] or Mobile Cloud computing [105] are a few examples of the current trend attempting to virtualize the resources provided by multiple surrounding devices in order to maximize resource usage and speedup computations. Liquid software shares the same goal to virtualize the capabilities of a set of heterogeneous devices in order to create a user experience in which an application can seamlessly run on multiple devices.

The technological foundation of liquid software emerges from the evolution IoT [6], the Web of Things [82], or more in general from the Programmable World [172]. Pervasive computing [157] shows how microprocessors can be embedded in all sorts of objects scattered around us. Today those *things* are not isolated from each other anymore, because they became "smart", and they are able to communicate with any similar object around them. The users and the devices surrounding them make up a complex ecosystem [186] which requires software to adapt to the set of available devices, for example whenever a smart object enters or leaves the proximity of the user running a given software application. Similarly, liquid software automatically flows between devices to adapt its deployment configuration to take full advantage of the resources and capabilities of multiple devices. Nowadays smart objects and devices are so common [182] and advanced [32; 79], that users may also interact with some devices that they do not directly own, but nevertheless they are allowed to share some information with it in order to run applications across particular devices. For instance, it is possible to find public displays [37] owned by cities [193] which may allow users to interact with them directly or by pairing their mobile smart phones with them. This way, users could for example take advantage of the large screen to display a picture slide show. This is a relevant scenario for liquid software, as the application should run across multiple devices to achieve the user's goal. More in general, the challenge we focus on in this section is how to enable devices to share their available computing resources and how to design software which

can seamlessly access them. However this new kind of interactions have a wide range of implications [8] that must be addressed, e.g., how can we trust a public display and how can we prevent data leakage [36]?

Mobile Computing [59] discusses the potential of creating powerful distributed systems made of mobile hardware that communicates through the Weeb. A mobile computing system trades portability and social interactivity with many distributed systems challenges such as dealing with device connectivity, discovery, trust establishment and proximity detection. The study of context-aware systems [160] allows us to understand how to create systems based on proximity-awareness [39; 129].

2.3.4 Mashups

This dissertation is also inspired by the multi-device mashups approaches [47], whereby Web applications are built out of the composition of reusable components. In particular, the UI is decomposed into Widgets, which encapsulate self-contained behavior and state [38]. Chudnovskyy *et al.* [35] 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 [34] 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. Daniel *et al.* discuss how to decomponse traditional Web applications into smaller reusable mashups components [46].

In the solution we implement in this dissertation we take a similar approach by using the HTML5 *Web components* standard [Moz19a], which provides reusable UI elements using standard Web technologies. With our approach we can also decompose solid applications, by turning them into liquid components.

Likewise, in [77; 81] Gaedke *et al.* research similar compositional approaches, define the composition model and discuss the specification of Smart Composition. Smart Composition [113] is a mashup tool that enables the development of multi-screen mashups. Smart Composition applies WebComposition to the simultaneous screening scenario. It is possible to roam from a device to another and exchange data between devices, but it doesn't provide solution for sequential screening or collaborative scenarios.

2.3.5 Distributed State in the Web

Liquid software requires data to be synchronized between a set of devices. Casteleyn *et al.* [33] discuss that the amount of state held by a complex Web application accumulated during a normal usage session cannot be easily migrated to another device. Likewise, it is also very challenging to synchronize such state among multiple Web browsers that access the same Web application concurrently on behalf of the same (or different) users.

InterPlanetary File System (IPFS) [Pro20] is a distributed, P2P file system. With IPFS it is possible to create distributed applications that process data files which are shared and synchronized among all the connected peers.

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 [148], devices discover one another via Bluetooth and the synchronization mechanism of the application state is provided by the centralized iCloud backend.

Liquid applications in a simultaneous screening scenario require to deal with replicated and synchronized data, which have been studied in the database community for many years [98]. On the Web, given the heterogeneity of the devices (e.g, the amount of storage may vary) and their frequent disconnections (e.g., due to battery or network connectivity problems), some of the assumptions of classical data replication mechanisms may need to be revisited.

2.3.6 Offload Computations in the Web

Web technologies have been evolving towards increased support for reliable mobile decentralized and distributed systems. In the past decade an effort has been made to improve and create new HTML5 standards [191] that can help with the creation of complex mobile distributed systems able to reliably maintain data synchronized between devices [147]. Thanks to novel standard protocols it is also possible to interconnect any device by using any standard compliant Web browser. Okamoto *et al.* [143] show how to create mobile Web distributed systems by exploiting the WebWorker HTML5 standard.

Web browsers allow any device to connect to a Web application, meaning that users can connect all the devices they own to run a single liquid cross-device

application. Whenever a device is connected to the liquid application, the resources it provides are exploited by the software. This approach is similar to the one of Volunteer Computing [3], where users willingly connect their own devices to perform a global computation, and share data, storage or computing resources among them.

Edge computing [164] focuses on optimizing data processing and storage by shifting computations closer to the source of the data, as opposed to shipping a copy of the data to large, centralized Cloud data centers [159]. The optimization reduces bandwidth consumption and latency in the communication between the edge devices, making it possible to reduce the overall processing time of an operation. Fog computing [21; 122] takes edge computing to the extreme, by making it possible to make all data processing computation within the IoT ecosystem. Liquid software also incorporates such performance goals, in order to seamlessly migrate applications among multiple user-owned devices without relying on centralized Web servers. Similar concepts can also be found in the ubiquitous computing [150] literature.

Traditionally large amounts of distributed computational resources was found mainly within clusters of computers or Cloud data centers, however recent trends show that also the Web, by employing Web browsers running across many types of devices and WebWorkers as a programmatic abstraction for parallel computations, can deliver a decentralized computation platform [42] as well. While most existing computational offloading work focuses on shifting workloads *vertically* from mobile devices to the Cloud [45], in this dissertation we study how to do so *horizontally* by using nearby devices. While these may not be as powerful as a Cloud data center, they will remain under the full control of their owners and enjoy a better proximity on the network.

Hirsch *et al.* [87] propose a technique for scheduling computation offloading in grids composed by mobile devices. The scheduling logic of the system is able to offload a set of heterogeneous jobs to any mobile device after an initial centralized decision-making phase. This is followed by the job stealing phase, in which jobs are relocated to other devices in a decentralized manner. The scheduler considers the battery status, the CPU performance and the up-time expectation of all connected devices when it has to decide where to offload jobs. The CPU performance is computed using a benchmark. While this approach shows promising results and it is able to increase the overall performance of computational-intensive applications, in this dissertation we present a fully decentralized approach able to operate inside a Web browser, where complete information about the devices hardware and software configuration is not always accessible.

Loke *et al.* [120], propose a similar system allowing multi-layered job stealing

techniques also with a hybrid approach (both centralized and decentralized) for offloading decisions. The decision depends on which devices are close to the device that starts the computation and then tries to scatter the job between them. Their approach is not based on a Web browser and relies on Bluetooth or WiFi for inter-device communication.

2.4 Industry Solutions

Today, perhaps the most illustrative example of liquid software is the *Handoff* capability (also known as *Continuity* capability) in Apple's iOS ecosystem [Gal14]. A typical Handoff use-case is a situation in which a person starts composing an email on an iPhone but then decides to finish the e-mail on a personal computer (Mac) that has a much larger screen and a physical keyboard. The participating devices need to be registered in the iCloud service with the same user identity, and the devices must be able to communicate over Bluetooth. When using Handoff, applications need to be written explicitly to take advantage of the Handoff API; furthermore, the applications must be pre-installed across all devices. Most of the built-in Apple applications are already compatible with Handoff, thus supporting continuity across devices. Each device runs a specific version of the application, and hence their UI is implicitly capable of adapting to take advantage of the device capabilities (e.g., multi-touch, screen size and specific screen resolution).

Another example of software that allows liquid-like user experiences on mobile devices is *Android Baton* from Nextbit [Kar14]; Baton runs on the Android ecosystem and provides features similar to Handoff. Thanks to the cloud backend it allows the users to synchronize any files between all the registered devices. It can also migrate the work the users are currently doing on a device to another one without losing the view that they had in the previous device. Baton has the same limitations as Apple Continuity: native Android apps need to be explicitly developed using this proprietary API to support migration and synchronization across devices.

Windows Continuum [Mic16] is a small physical device (box) that can be connected to mobile devices running Windows 10. Once a mobile device is tethered to the box, Continuum can then be attached with a cable to screens, keyboards and mouses around the user using it. Any hardware attached to the Continuum box is interfaced with the mobile device, making it possible to seamlessly migrate from the mobile view on the device to a desktop-like experience on the attached screen. With Continuum there is no need to use a cloud service for synchronizing

data nor to have communication between multiple devices; the box itself reads the data from the mobile devices and maps the view displayed on the phone with a responsive desktop view that is displayed on the connected screen. From the users' point of view the work they are doing is migrated from the mobile device to the screen. Google Chromecast [Goo20a] uses the same concepts as the Windows Continuum; thanks to external hardware Chromecast can interface multiple devices with a television, allowing migration of supported applications into an external screen.

Opera Unite (now discontinued) [Ope09] was a Web browser extension which allowed users to host social Web applications (e.g., photo sharing, social wall) on their Web browser. The goal of these efforts was to enable safe social networking whereby personal data would be exchanged directly between trusted devices.

2.5 Cloud

Cloud computing systems, thin client environments and *Web-based applications* that adhere to the Software as a Service (SaaS) principles [171; 25] naturally possess many of the qualities required by Liquid Software. For instance, since in cloud-based applications the majority of the data resides in the cloud, in principle all the clients using the same application will stay in sync automatically. However, in the absence of mechanisms that notify the clients of changes made by other clients, in practice all the locally stored or cached client-side data will quickly go out of sync. In desktop-based systems that utilize the Web browser as the client environment, these problems are usually mitigated by explicitly reloading the page containing the Web application. In mobile Web computing environments such as Firefox OS [Moz12] (popular in 2012 and now discontinued) or Cloudberry [176] explicit use of notification mechanisms (e.g., push notifications) is required at the application level if multiple clients are to be kept in sync.

Cloud-based systems such as Apple's iCloud [App18] and Google Sync [Goo20b] are already paving the way for automatically synchronized devices. However, these systems are limited to devices supporting the same native ecosystem. Furthermore, these systems do not yet provide seamless experiences and transitions across devices. Ideally, when the users move from one device or screen to another, the users should be able to continue doing exactly what they were doing previously, e.g., continue playing the same game, watching the same movie or listening to the same song on the other device. This type of truly liquid usage of software applications is not generally supported yet,

although such features may already be available at the application level. For instance, the Spotify music player allows the user to shift the currently playing music track from one device to another.

In the literature there are many other frameworks that enable the creation of Web applications with some of the requirements originally described in the liquid software manifesto [174]. For example PolyChrome [13] is a centralized Web framework for building co-browsing applications, where the implemented views can span on multiple surfaces deployed on multiple devices. The framework defines and supports four predefined layouts: *stitching*, *replication*, *nesting*, and *overloading*. The PolyChrome API makes a distinction between *interactions* and *events*: *interactions* change the data of the application and are sent to the Web server where the central state of the application is stored; *events* change the view displayed on the devices and are directly exchanged between all paired devices. Polychrome can create components out of legacy applications in order to create views spanning across multiple devices. The design of Polychrome is strongly centralized, which does not allows users to define which of their data is stored on the Web server. In Chapter 3 we present more related liquid frameworks as we discuss the design of liquid software.

Chapter 3

Liquid Software Design

3.1 Design Considerations

The creation of the seamless user experience able to transparently flow from a device to another requires to follow several design considerations. Consequently the effort and complexity of the underlying implementation of liquid software increases as the user experience needs to follow more principles. In this section we discuss individual architectural considerations that entail a number of key design decisions that form the design space of liquid software. From the design considerations we derive the design dimensions that are later discussed in the design space (more about it in Section 3.2).

The core design considerations important to liquid software are: • User Interface (UI) adaptation; • data and state synchronization; • client/server partitioning; • security. All design considerations must be taken into consideration by software developers during the design of their liquid software.

3.1.1 User Interface (UI) Adaptation

The core feature of liquid software is its ability to adapt to take advantage of each and every device. The **device usage** of liquid software is the first design dimension that developers must consider during the design of their liquid applications. Liquid software design can be different depending if the liquid software runs sequentially or in parallel on the set of devices. The device usage dimension does not distinguish between collaborative or single user scenarios.

The LUE of a liquid application needs to consider and must be able to detect and operate with all different *input methods* provided by the devices. E.g., the LUE must be able to operate with either a keyboard or a touch screen depending

on the deployment. Depending on the nature of the device, it is possible that the LUE must display only a subset of the data of the application, e.g., a small device might show only the most meaningful data as opposed to a device with a larger display. When multiple devices are available, the use of *companion devices* makes responsiveness more challenging since the UI needs to adapt to a combination of complementary devices. Thus, in general, the UIs of liquid applications should be responsive and adapt to the set of devices where they currently run. The level of adaptation of the LUE can be summarized with the **UI adaptation** design dimension.

While traditional software, meant to run on a single device, expects that both input and output channels operate on the same device, liquid software does not have such restrictions. Liquid applications may allow multiple input channels as well as multiple output channels from different devices. The users are free to use any number of their devices as they please, making it possible to use the devices as comfortably as possible, e.g., by using the keyboard attached to a computer to type on the smartphone. The only assumption is that the liquid application has been deployed on all devices. In the simplest case, this could be achieved with basic input/output forwarding, without the need to actually migrate, fork or clone the software. The kind of input/output interactions of liquid software can be summarized with the **LUE primitives** design dimension.

3.1.2 Data and State Synchronization

When we talk about software able to dynamically change its deployment at runtime it is important to distinguish between *persistent application data* and *dynamic application state* [55]. By persistent data we refer to the static content (e.g., documents, images, media files) that the users store persistently across usage sessions, while dynamic state is the runtime information that the application needs during its execution.

In existing designs, the persistent data is commonly stored locally on each device and can be synchronized using different cloud-based storage services [108]. Unfortunately, many of the cloud-based storage systems are limited to individual applications, specific data types or certain native operating system ecosystems (e.g., Apple's iCloud [App18] or Google Sync [Goo20b]).

Liquid software is also concerned with the dynamic state of the application. The runtime execution state can be captured at different levels of **granularity** (another design dimension of the liquid software design space), from the values of relevant properties (e.g., storing UI configuration settings) or the entire volatile memory storage of an application (e.g., the JavaScript (JS) Heap that

contains all stored objects in a Web page). In traditional solid software applications, the state is not persisted after an application is turned off. In liquid software, the seamless LUE requires to decide how the **data is replicated** and where it is **persisted**, regarding the state it is important to **identify** which parts are bound with the migration of the applications, and how it is **synchronized** across the devices.

3.1.3 Client/Server Partitioning

The partitioning between the client and the server is a key design decision for liquid architectures [126], especially when we consider to implement liquid applications with Web technologies, for this reason we introduce the **layering** design dimension. Applications can be meant to run entirely on a backend server; the client-side of the application can just display the UI and delegate the processing of all the events to the server (e.g., Ruby on Rails [Rai16]). Similar ultra-thin techniques exist on native liquid applications where a virtual desktop is offered for remote use (e.g., in SunRay terminals [Ora16]). In contrast with the server-centric view, the client-oriented approach focuses to run thicker clients. Originating from Asynchronous JavaScript And XML (AJAX) [65], we today have single-page Web applications that use Backend as a Service (BaaS) APIs such as Firebase [Goo20c] deployed on a central server, providing persistent storage and notification services that are shared among all clients. In the realm of native clients, the most obvious approach is to use reflection for transmitting the state of a Java application from one virtual machine to another, as originally proposed in the context of liquid software by Joust [85].

The development liquid applications can be both client- and server-centric at the same time, as it is possible that applications are deployed both on thick clients while partially depending on the logic deployed on the server-side. E.g., Vaadin framework [78] or Google Web Toolkit (GWT) [Goo16a] allow the development of powerful UIs components, while the focal point for developers is on the server side of their applications. In the context of liquid applications, the balance between centralization and decentralization can even change dynamically. Different devices have different capabilities, and thus optimal configurations may vary. Therefore, liquid software frameworks should offer capabilities for offloading computation from clients to servers and vice versa. Since the capabilities of computing devices may vary considerably, we anticipate a full range of architectural choices from ultra thin (all computations performed in server side) to ultra thick (clients are completely self-contained) solutions. The different level of centralization and decentralization can be summarized by the **topology** design

dimension.

3.1.4 Security

The users should generally remain in full control over dynamic deployment and transfer of applications and data. If certain functionality or data should be accessible only on a specific device, the user should be able to define this in a simple, intuitive fashion. E.g., in SunRay terminals the user's session was secured with a smartcard that the user had to enter in the terminal in order to open the session [Ora16]. Similarly, when an application migrates to the devices owned by foreign users, either belonging to other users or shared public devices, suitable access control policies need to be established and enforced. While security aspects are often downplayed for software running on multiple devices belonging to the same user, there is a need to assess and evaluate to which extent existing security solutions can be applied to liquid software.

3.2 Design Space

A LUE can be implemented in a number of different ways. The design space of liquid software arises from issues and choices in replicating and synchronizing the software components and their state, and there can be various motivations behind the design decisions. For instance, the users who switch the device in the middle of a task do not appreciate if they have to restart their work from scratch; rather, they expect continuity in the hand-off of the work between devices, including seamless availability of their data [Gal14]. It is important to discuss whether such synchronization relies on a centralized or a decentralized architecture. In a centralized architecture all the software components and their state are backed up in the cloud, and the devices synchronize their state via centralized servers. Alternatively, in a decentralized approach liquid software flows directly between devices, leveraging P2P connectivity for direct state synchronization across devices. The granularity of the software components that need to be migrated also impacts the LUE, especially when deploying a liquid Web application over multiple devices that are intended to be used at the same time.

To sketch the design space for liquid software, we will next discuss the relationships and dependencies between a number of design issues and alternatives (see Figure 3.1). The design space model can be read from top to bottom following the relationships between the various alternatives in the design space. Some alternatives are exclusive (e.g., the different levels of **granularity**), while others

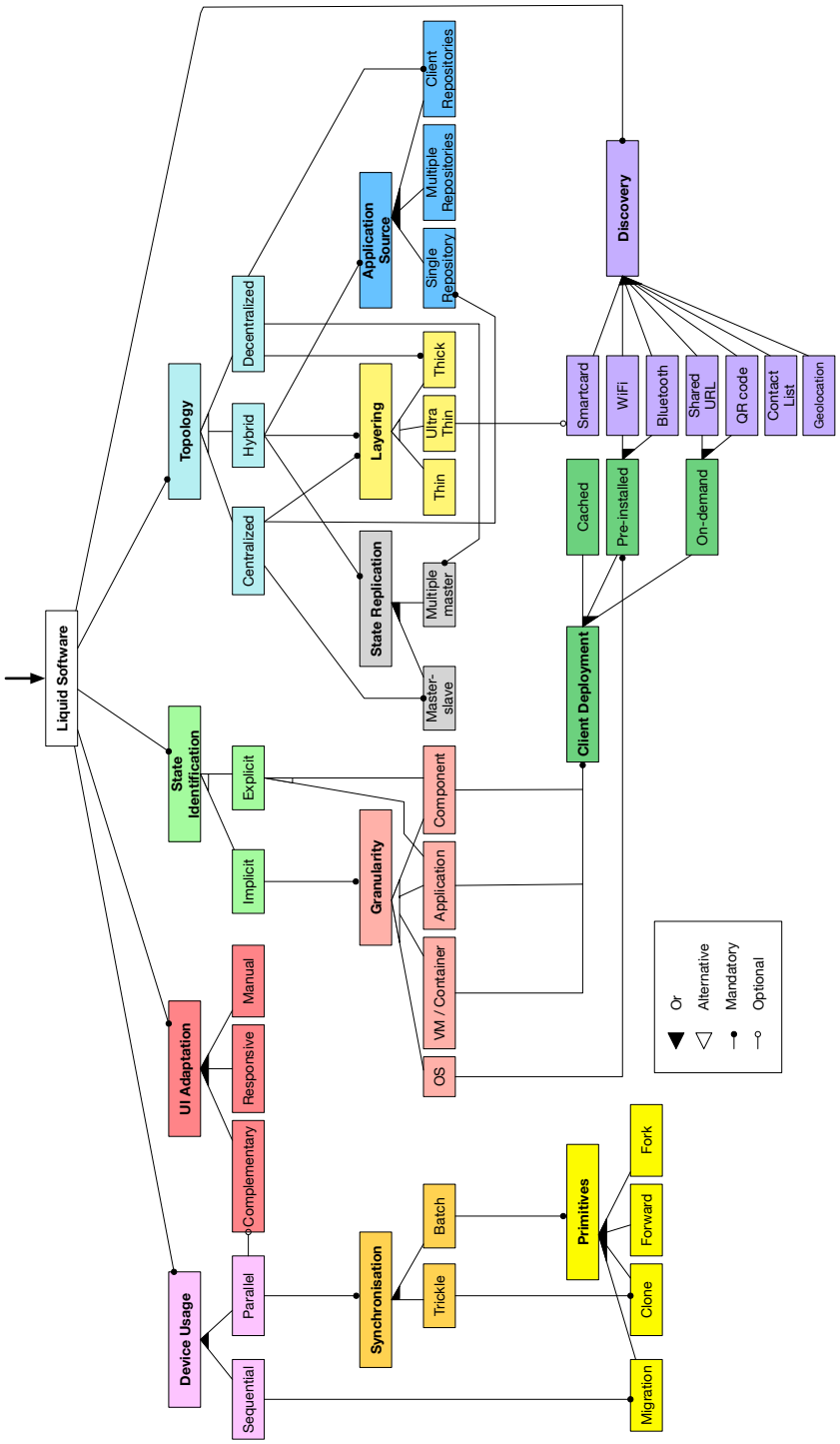


Figure 3.1. Overview: the design space of liquid software. Mandatory arrows indicate that a child feature is required; optional arrows indicate that the child feature is optional; alternative arrows indicate that only one child feature must be selected; or arrows indicate that at least one child feature must be selected.

can be selected together (e.g., which LUE **primitives** are supported). We also indicate how the different alternatives constrain each other.

Table 3.1 characterizes and positions different technologies within the design space. The goal is to provide proof-of-existence for each design space alternative by citing concrete technology examples supporting it. However, we do not intend to present a complete survey/review of existing technologies for liquid software.

3.2.1 Topology

The *topology* of a liquid architecture can be *centralized*, with a single, well-defined host that maintains the master copy of the application state and an image of the software to be deployed and run on each device. This centralized host is usually available in the cloud, taking advantage of the high availability and virtually unlimited capacity of data centers, potentially at the expense of the privacy of the data that is no longer confined only to user-controlled devices. Liquid software thus flows up and down from the cloud onto various user devices that are thereby implicitly backed up and synchronized as long as a connection to the cloud is available.

Alternatively, liquid software architectures can be designed with a *decentralized* topology in which software, the state of the applications and their data are exchanged directly between devices in a P2P fashion, leveraging local connectivity between devices. While P2P approaches can work by restricting the deployment of software onto specific devices that are under the user's control, such a *multi-master* approach (as opposed to centralized *master-slave* approach) makes it more challenging to resolve synchronization conflicts since there is no single master copy. Furthermore, while an individual device may have perfect Internet connectivity, it is unlikely that all the user's devices are always online at the same time. Thus, special care must be taken to ensure successful migration and synchronization of state across all paired devices. Conflict handling can become especially problematic if a device has been used actively in offline mode for a long time (e.g., during long intercontinental flights).

This basic topology decision, centralized versus decentralized design, can be regarded as a fundamental dimension in the context of liquid software. Granted, with a central server, it is easier to manage software as well as data content. However, the decentralized alternative can offer significant benefits as well, since only local connectivity is needed for migrating state from one device to another, and the user's data can be kept outside the reach of major cloud providers. Hybrid approaches are possible too, with the cloud serving as an additional *peer*, e.g., for backup purposes; a similar approach was used in Nokia's EDB system [108].

Table 3.1. Technologies positioned in the liquid software design space.

		Sun Ray [Ora16]	Joust [85]	Fluid Computing [22]	Apple Continuity [Gal14]	Android Baton [Kar14]	Cloudberry [176] Cloudbrowser [173]	Continuum [Mic16]	XD-MVC [89]	Liquid.js for DOM [179]	Liquid.js for Polymer [66]
		1997	1999	2005	2014	2014	2014	2015	2015	2016	2016
Architecture	**Topology**										
	Centralized	✓			✓	✓	✓				
	Decentralized			✓							
	Hybrid		✓					✓	✓	✓	✓
	Application Source Topology										
	Single repository	✓	✓		✓	✓	✓		✓		
	Multiple repositories							✓		✓	✓
	Client repositories			✓							✓
	State Replication Topology										
	Master-slave	✓			✓	✓	✓				
	Multiple masters		✓	✓				✓	✓	✓	✓
	Layering										
	Ultra Thin Client	✓						✓			
	Thin Client						✓				
	Thick Client		✓	✓	✓	✓			✓	✓	✓
	Client Deployment										
	Preinstalled	✓	✓	✓	✓			✓			
	On-Demand					✓			✓	✓	✓
	Cached						✓			✓	✓
	Granularity										
	OS	✓						✓			
	VM/Container										
	Application		✓	✓	✓	✓	✓			✓	
	Component						✓		✓		✓
State	**State Identification**										
	Implicit	✓						✓			
	Explicit		✓	✓	✓	✓	✓		✓	✓	✓
	Synchronization										
	Trickle	✓	✓	✓			✓	✓	✓	✓	✓
	Batch			✓	✓	✓	✓			✓	
Liquid User Experience	**Device Usage**										
	Sequential	✓		✓	✓	✓	✓		✓		✓
	Parallel		✓	✓				✓	✓	✓	✓
	UI Adaptation										
	Manual		✓	✓					✓	✓	✓
	Responsive	✓					✓		✓	✓	✓
	Complementary				✓	✓	✓	✓	✓		
	Primitives										
	Forwarding	✓	✓					✓	✓		
	Migration				✓	✓	✓			✓	✓
	Forking								✓	✓	✓
	Cloning			✓			✓		✓	✓	✓
	Discovery										
	Shared URL		✓				✓	✓	✓	✓	✓
	QR Code								✓	✓	✓
	Bluetooth			✓	✓	✓					
	WiFi			✓		✓					
	SmartCard	✓									
	Contact List								✓		
	Geolocation								✓		

3.2.1.1 State Replication Topology

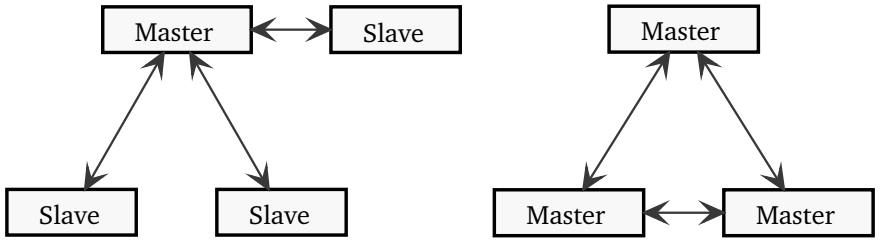
In real-life implementations, the borderline between the two basic topologies is not always clear. For instance in [106], implementation techniques forced the design to use a centralized server for communication, while conceptually migration was handled in a distributed fashion. It should also be noted that the sharing of the user's data, synchronization of the application state, and application deployment do not need to be organized according to the same topology; the final architecture may be a mixture of centralization and decentralization. Thus, we further distinguish *state replication topology* from *application source topology*. More specifically, the state replication topology (see Figure 3.2) alternatives consider:

- **Master-slave** (see Figure 3.2a): state replication is centralized. There is one single master, such as the cloud or a Web server, which owns the master copy of the state and makes sure that all connected clients (slaves) receive consistent updates. Each time a client updates the state it must communicate with the master; the master can drop the request or accept it; in the latter case the update is propagated to all the other slaves. The master-slave approach can be burdening for the node acting as a master because all the requests are managed by a single node.
- **Multiple masters** (see Figure 3.2b): state replication is decentralized. All the clients act as masters, which discard or accept state changes and propagate them to the other clients that need to agree with them. In case of multiple masters, conflict resolution becomes more challenging as it requires the implementation of a suitable distributed consensus protocol [98].

3.2.1.2 Application Source Topology

In the area of application source topology (see Figure 3.3) we recognize the following alternatives:

- **Single Repository** (see Figure 3.3a): the master copy of an application is stored on a single node such as a server in the cloud or a Web server. The single repository structure is the simplest to implement; whenever the liquid application is requested, clients will look for it in this node. As new versions of the application are released, it is sufficient to replace the master copy of the application with the new one. Moreover, the single repository also stores the dependencies of the application that can be retrieved together with the application.
- **Multiple Repositories** (see Figure 3.3b: the master copy of an application is stored in multiple nodes, such as multiple Web servers. In the multiple repos-



(a) Master-slave (b) Multiple masters

Figure 3.2. State replication topology alternatives.

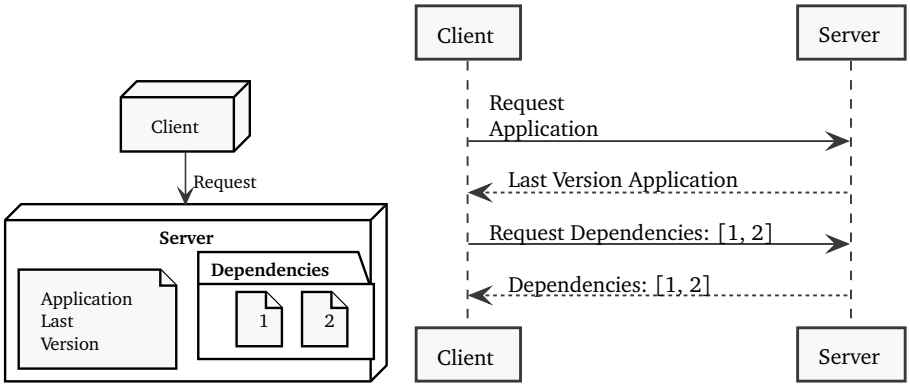
itory structure, the master copy of an application can be replicated and stored on multiple nodes, and the application and its dependencies can be stored separately from one another. As new versions of the application are implemented, they must be pushed/propagated to all the repositories (e.g., using a content delivery network). In the case of full replication of the nodes, it is possible to retrieve an application even if one the nodes fails, because another node can provide the application on its behalf.

• **Client Repositories** (see Figure 3.3c: in this option the clients store the application and can share it with the other clients. This solution can be implemented in decentralized topologies if the clients are able to communicate with each other through P2P communication [181]. In this alternative it is difficult to manage versions of the application as they are pushed to clients, because two clients could be running different application versions. In this case clients should be able to recognize if they are using the same version of an application and update the newer one if they are not.

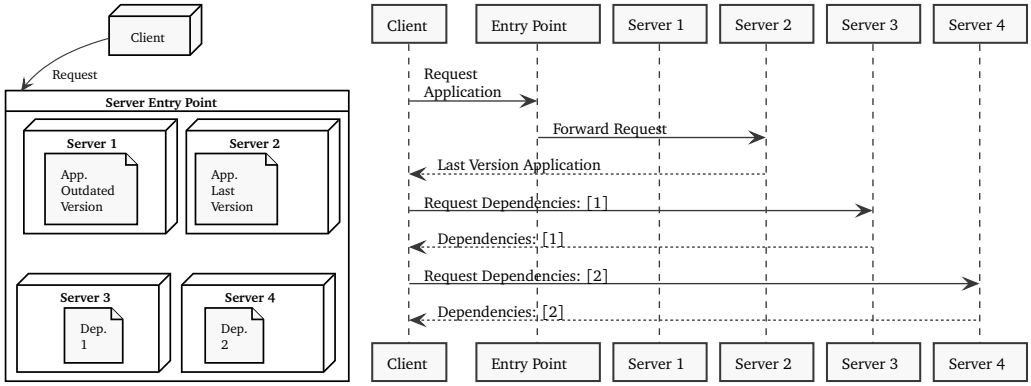
The selection of topology depends also on the expected user experience behavior when dealing with temporary device outages and offline scenarios. When the user is moving sequentially from one device to another, there might be significant gaps between executions, e.g., if the target device is not online when the previously used device has been switched off. A centralized topology can introduce a store-and-forward functionality that allows migration in sequential usage scenarios despite the temporary unavailability of some devices.

3.2.2 Discovery

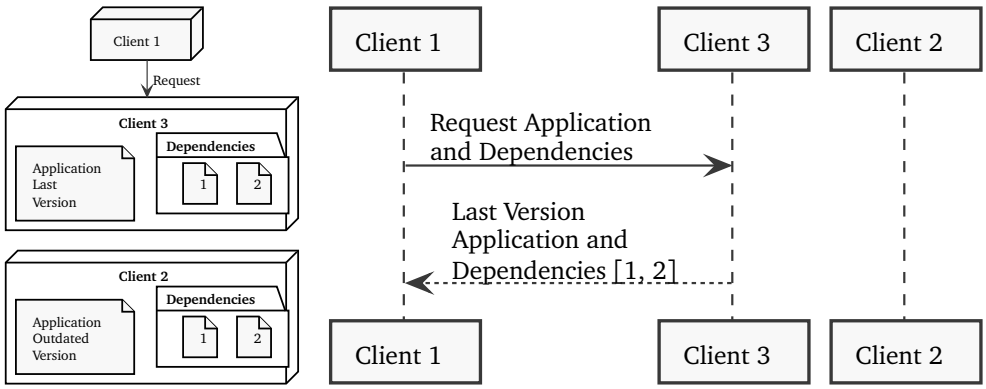
An important aspect is to define how liquid software becomes aware of the set of target devices on which it can run. The *discovery* mechanisms are concerned with the existence of the devices, their location/proximity, their current reachability (online/offline) and their ownership. In centralized topologies, the registry



(a) Single repository – deployment and process views.



(b) Multiple repositories – deployment and process views.



(c) Client repositories – deployment and process views.

Figure 3.3. Application source topology alternatives.

of devices is usually kept in the cloud. On the device side, several technologies are readily available for discovery, including *shared Uniform Resource Locators (URLs)*, *Quick Response (QR) codes*, *Bluetooth* service discovery mechanisms, *Wi-Fi* access point connectivity, and special purpose hardware such as *smartcards*. Discovery can also be based on social interactions between the users, e.g., by employing *contact lists* that connect the devices that the user wants to use together.

Existence Discovery: identifying all the available devices is the minimum requirement any liquid software system has to fulfill in order to enable a LUE. Many techniques can be employed in order to make the liquid software system aware of the available devices: by creating a Local Area Network (LAN) or Personal Area Network (PAN) whenever access to the Internet is not necessary or possible (e.g., Wi-Fi or Bluetooth); or by accessing the same Web server and communicating through the Internet when a wider network is needed (e.g., shared link or QR codes). In the former case the devices can be identified by their Media Access Control (MAC) addresses, while the Internet Protocol (IP) address can be used in the latter case. In either solution the user has to connect to a shared network and know in advance the access point or the server's URL. The less configuration setup operations the user has to perform, the better. Some solutions, such as scanning a QR code, hide the complexity of entering long URL addresses from the user, while in the Wi-Fi scenario, the discovery can be transparent if the application is configured to automatically connect to a default access point whenever its Service Set Identifier (SSID) is detected.

Location Discovery: location discovery focuses on *locating* the relative position of all the connected devices with respect to each other. The relative location information is not a strict requirement but it can highly enhance a LUE. For instance, by knowing the relative position of two devices, it is possible to know the direction and distance between the two, making it possible to support specific gestures for migration, forking and cloning. A notorious example in this area was the Microsoft Surface Table [Mic17] (nowadays discontinued) prototype that allowed phones to share pictures as they were placed on top of the table display. A liquid software system built on top of popular LAN technologies such as Wi-Fi or Bluetooth can easily compute the relative location of the connected device by using fingerprinting algorithms [117]. A liquid software system built on top of the Web can use more complex geolocation technologies such as GPS that are more energy consuming compared to Received Signal Strength Indication (RSSI)-based approaches [104].

Ownership Discovery: ownership discovery focuses on *assigning users to devices*. It is critical to ensure that only authorized software can run on a device and that the users can control where their data is replicated to. Devices belonging

to the same user can have a higher level of trust than devices temporarily paired between different users. Some devices (e.g., public displays) may be shared temporarily among multiple users (e.g., linked by a given social networking relationship); for quite obvious reasons, no information should be automatically replicated to such devices. Ownership discovery requires the users to authenticate their identity on each device. This may happen in a number of different ways: with a passcode, a user/password login prompt that is verified by a third party, a shared secret among all devices (which can be propagated along using QR codes), a smart card, or a combination thereof.

3.2.3 Layering

Today, the majority of Web applications include both server and client (end-user device) layers. There are multiple ways to split the application between the server and the client. Applications that perform the majority of their computation on the client side are known as thick client applications, or more commonly *rich client* applications [33]. Applications in which the vast majority of computation occurs on the server side are known as *thin client* applications. There are even extreme *ultra thin* approaches (e.g., SunRay [Ora16]), in which the primary function of the end user device is to render pixels, only acting as a remote display and terminal to access software that is otherwise run entirely on the server. In thick client applications the majority of computations run on the client, and the server's role is usually limited primarily to data storage.

Naturally, there is a full spectrum of architectural alternatives between purely thin and thick client architectures. In Figure 3.4, we enumerate different logical layers of a Web application designed according to the Model-View-Controller (MVC) pattern [121]. While thin clients only run the view layer, thick clients may run all the layers or only leave the Model to be handled by the server. The maturity levels of the maturity model of liquid Web applications discussed in Section 3.3 depend on the layering of a liquid application.

The typical criteria [45] and trade-offs for selecting between thin and thick client architectures include the following:

- *Computing power*. While servers typically have more powerful CPUs and more memory, these resources may be shared by several users. The more limited (but potentially still substantial) resources available on clients are usually dedicated to one user only.
- *Battery and energy consumption*. The users care about the length of the time they can use their devices between charging. The less computation is done in battery-operated client devices, the longer the batteries can generally be

expected to last. However, since it is often *network traffic* that dominates power consumption, overall battery life sometimes improves considerably by performing more computation on the clients.

- *Perceived performance*. The users typically enjoy highly interactive applications. Frequent network requests may cause delays. To improve the perceived performance of Web applications, technologies such as AJAX [65] and single-page applications [131] have emerged.
- *Required bandwidth*. Available bandwidth is one of the key considerations in driving and defining practical use cases for liquid software. The longer it takes to migrate the execution from one device to another, the less appealing it will be to use the mechanisms supporting multi-device usage.
- *Offline operation*. Thin client applications are typically unusable if the network connection is down, while thick client applications may continue their execution even without active network connection.
- *Direct hardware access*. Thin client applications that run in a sandbox often have limited access to the capabilities of the underlying local runtime environment. In contrast, thick client applications can usually directly utilize local hardware resources such as cameras, sensors, Graphics Processing Unit (GPU), and the file system.
- *Engineering challenges*. Applications whose computation and data are distributed between the client and the server are more difficult to develop and maintain than applications that are deployed only on the client or on the server [126].

In the context of liquid applications, the balance between server and client execution can even change dynamically. Heterogeneous devices may have highly divergent computing capabilities, input mechanisms and other resources, and thus optimal configurations may vary. Therefore, liquid software frameworks should offer capabilities for dynamically migrating computation from servers to clients and vice versa.

3.2.4 Granularity

While the majority of use cases for liquid software are concerned with the migration of entire software applications, we have recognized a variety of use cases that call for liquidity at different levels of granularity. In the following we show which layer(s) of the software stack can be made responsible for migration and synchronization (see Figure 3.5).

- *Operating system level*. The operating system and its underlying resources such as the file system, communication middleware and UI follow the liquid soft-

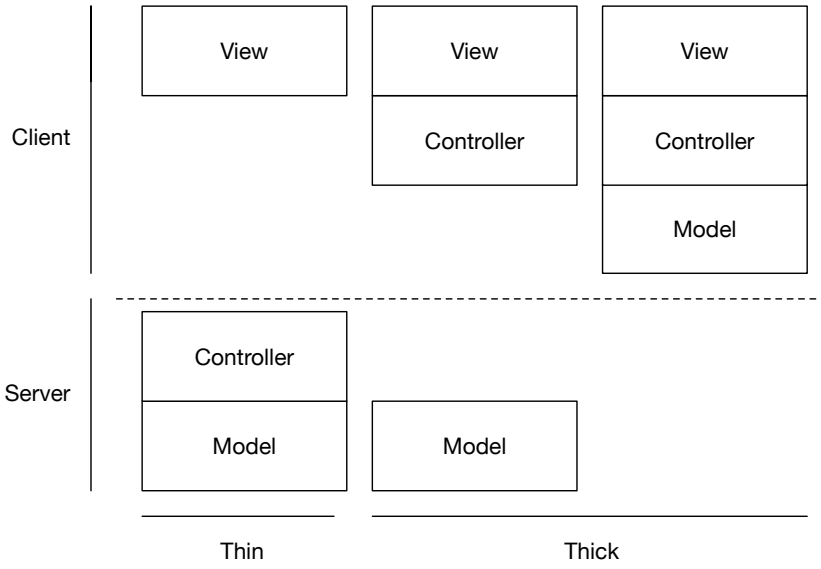


Figure 3.4. Layering alternatives

ware principles. Technically this means that operating processes can fork and migrate across different devices, state synchronization is seamless and all the data is automatically available to all devices. For the end user this means the liquidity is not limited to specific applications and that all the applications are liquid by default. Implementing liquid software at the OS level is the most comprehensive but also the most complex approach since it needs to deal with hardware differences, security, resource consumption, live process migration, and various other issues. One obvious limitation is that all devices participating in the LUE need to run the same operating system.

- *Virtual machine/Container level*. Probably the most commonly used mechanism for migration today is to utilize Virtual Machines (VMs) that enable the transfer of running applications between various computing devices. The technology is widely used in data centers, e.g., to bring applications and content closer to the edge of the network, and consolidate multiple VMs to run on the same physical resources to save energy. Like VMs, containers are widely used in cloud systems, with the advantage of reduced footprint and more fine-grained control on which parts of the system can be migrated. While limitations are also similar between the two approaches above, in the context of containers problems related to bandwidth can be at least partially solved by carefully selecting the parts of the system that must move.
- *Application level*. Moving a specific application as it is running is probably

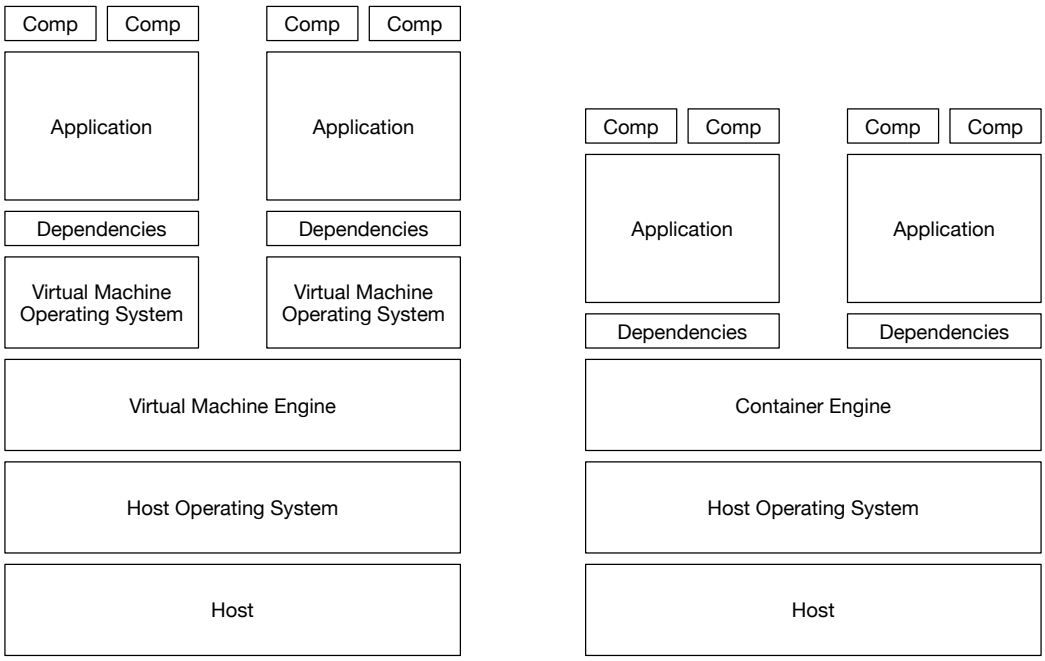


Figure 3.5. Granularity alternatives

the most natural way to consider migration; application developers are commonly offered a framework that they can utilize for implementing state synchronization at the application level. The framework may offer capabilities that the developers can use to control which parts of the state and data are migrated.

- *Component level*. Migrating application components from one device to another enables custom and flexible designs, where only parts of applications that need to be present in the target device are transferred. This level of granularity becomes especially interesting in companion scenarios in which multiple devices are used at the same time. This can be an efficient way to implement the *complementary* screening scenario in which different devices are used for presenting different visual components and controls of the same application.

Design decisions related to granularity are heavily dependent on the capabilities of target devices. For instance, with ultra thin clients only the visual presentations (in the extreme case only "pixels") need to transferred to the target client. In contrast, a thick client typically requires at least application level liquidity support.

3.2.5 Client Deployment

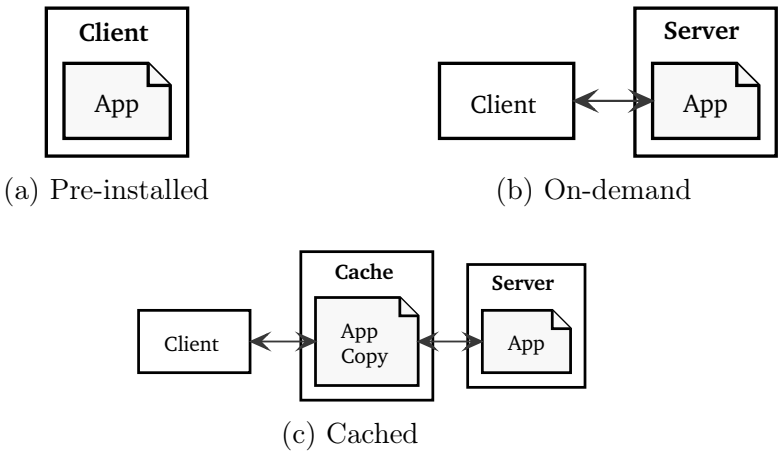
There are numerous different ways to implement client software deployment (see Figure 3.6) and installation. At one end of the spectrum there are *preinstalled* applications that are statically installed, similarly to the applications in personal computers. This method is used for native applications in major mobile platforms such as Android, iOS and Windows Phone. Even Web applications in some platforms, such as Tizen [Lin16] and Firefox OS [Moz12] follow the same paradigm: the applications are prepackaged, transferred to the device (often by downloading them from an application store), and then installed in the traditional fashion. On many of the current native mobile platforms, a cloud service (e.g., iCloud) will automatically (and entirely transparently from the user's viewpoint) install previously acquired applications when the user takes a new device in use.

At the other end of the spectrum there are *on-demand* Web applications that are run simply by pointing the Web browser or Web runtime to a specific URL. These applications are typically downloaded on the fly for each execution, and are only available in the presence of a network connection. In such systems code deployment means nothing more than passing on the URL of the application from one device to another, giving access to a server running on premises, in the Cloud, or on a hub installed in the smart home of the end-user.

In Cloudberry HTML5 mobile phone platform [176], applications were run by giving the URL to the Web engine; the application code was then *cached* using the HTML5 Application Cache [Moz19b]. The application cache would keep the necessary files available so that dynamic code downloads were subsequently needed only if some of the implementation components of the application actually changed. Nowadays the Application Cache is deprecated, but the new HTML5 Service Workers [Moz20b] could be used instead.

Although the deployment mechanisms are technically independent of each other, there are some logical connections. The following combinations can be encountered commonly in real-life implementations (see Figure 3.6):

- *Thin client, on-demand deployment (see Figure 3.6b)*. For thin client applications offline operation is not necessary and thus on-demand deployment is a feasible option.
- *Thin client, pre-installation(see Figure 3.6a)*. In thin clients the majority of functionality resides on the server; application updates are also server-driven. In many frameworks the client application is generated dynamically and may change in response to changes on the server side.
- *Thick client, on-demand deployment (see Figure 3.6b)*. One of the main



(a) Pre-installed (b) On-demand

(c) Cached

Figure 3.6. Code deployment alternatives

benefits of thick client applications is the built-in support for offline use when network connection is not available. In Web applications, this benefit can only be achieved with Service Worker (or Application Cache for old deprecated applications).

• *Thick client, pre-installation (see Figure 3.6a)*. This combination resembles the traditional, native, installable binary applications. Obviously, offline use of such applications is possible by default unless the application logic itself relies on network connectivity.

In the extreme ultra thin systems there is no application installation to end user devices at all. Rather, all the installations take place on the centralized server. Conversely, in ultra thick designs, especially those leveraging P2P synchronization, the server might not be needed at all since everything is managed by the clients themselves.

3.2.6 Liquid User Experience (LUE)

True LUE consists of two parts: primitives that are to be supported, and adaptation techniques that are applied when an application is moving from one device to another, where the characteristics of the device are different. These will be discussed next.

3.2.6.1 Primitives

From the user's perspective, on an individual device liquid software acts just like any other software. However, in order to create a seamless user experience

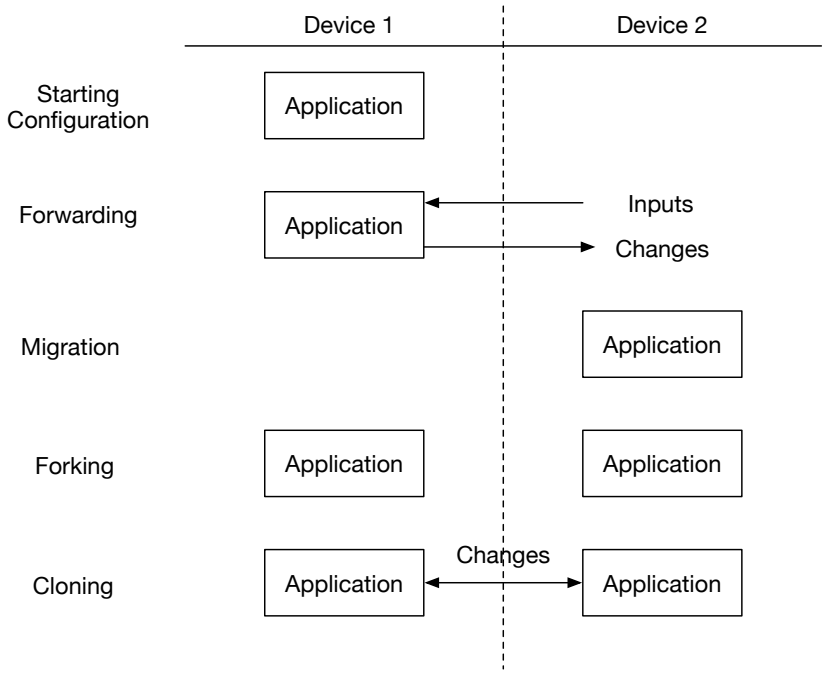


Figure 3.7. LUE primitives

reflecting the mobility of software [55] from one device to another, a combination of the following four primitives is used (see Figure 3.7):

• *Forwarding*: the ability to transparently forward the output and redirect the input gathered on one device to the application remotely running on the other device.

• *Migration*: the ability to partially or completely move the current instance of the liquid application from one device to another effortlessly.

• *Forking*: the ability to partially or completely create a copy of the current instance of the liquid application on a different device.

• *Cloning*: the ability to partially or completely create a copy of the current instance of the liquid application on a different device (i.e., forking) *while keeping the two instances synchronized thereafter*.

According to the liquid software Manifesto [174], the user is supposed to remain in full control of where the software is running: *forwarding*, *migration*, *forking* and *cloning* primitives allow the user to roam from a device to another. The *migration* primitive is used mainly in sequential screening, enabling the single user to move the liquid application among the user's own devices. This establishes continuity in the use of the application across multiple devices; for ex-

ample, when the user is watching a movie on the phone during a daily commute, the movie will continue playing from the same position on the large screen TV when the user arrives at home.

The *forking* and *cloning* primitives are more suited for parallel and collaborative screening scenarios, where the state of an application must be shared among many users or devices. This establishes a complementary, companionship role among multiple devices that are used at the same time. For example, a user going through a checkout process on an e-commerce Website accessed via the desktop Web browser may simultaneously use the secure fingerprint reader of his smartphone to validate the ongoing credit card payment transaction.

3.2.6.2 User Interface (UI) Adaptation

There are three different possible alternatives for deciding how to perform the UI adaptation to the set of devices that are used for running the application [116]: *manual*, *responsive*, and *complementary*. With a manual approach, the users may directly activate the LUE primitives to control how the UI is deployed onto devices. From the developer's perspective, the manual alternative requires the development of `N` versions of the application, one for each device targeted by the deployment. While this is a common practice for mobile smartphone platforms, the costs of this approach for further growing the supported number of platforms and device types could become prohibitively expensive.

In contrast, a responsive design is used for adapting the same application software to the device's features such as its screen size. It adjusts the UI by considering the different input and output capabilities of the target devices. For example, a small device might show only the most meaningful data as opposed to a larger device that would display the full contents. Existing mechanisms and design practices such as *responsive Web design* [125] pave the way to automatically treating this dimension, although still requiring careful attention and consideration from the UI designers and application developers.

Overall, liquid software can fill all the available devices and provide not only a responsive user experience (where the UI is adapted to each device's capabilities), but also a complementary user experience (where the capabilities of all the devices are fully exploited by the application with a distributed UI).

It must be kept in mind that liquid software behavior is always to some extent an illusion – a lot of technical grunt work is often needed under the hood in order to maintain a seamless user experience and the users' impression that software is truly "flowing" across devices. For instance, in many cases the developer may use pre-rendered bitmaps instead of constant repainting in order to create an

impression of smooth application transfer. A significant part of the designers' and developers' work is concerned with maintaining such an experience.

3.2.7 Data and State

Liquid software systems deal with two kinds of data: 1. persistent user data and 2. ephemeral runtime application state. Persistent user data needs to be made available across different devices and usage contexts. Likewise, the ephemeral, dynamic state of running applications must be stored in a form that allows the state to be effortlessly migrated or synchronized across devices, either fully or partially. The *state identification* can happen *implicitly*, where all parts of the application are addressed, or *explicitly*, where only relevant parts are synchronized.

Conflict handling and consistency. Different user experiences impose different requirements on state synchronization. In *sequential screening* there are no conflicts, since there is only one active device at each time. In contrast, in parallel and collaborative uses, there is the necessity to synchronize the state in real-time and it may lead to conflicting updates to the same state. In general, if multiple devices are active at the same time, conflicts between their states may become an issue. Some of these problems need to be solved in the application level, but ideally the underlying application or OS framework should guarantee the eventual consistency in data synchronization.

At the implementation level, state synchronization can take place in two different ways: *trickle* and *batch updates*. In the former case, two or more devices are kept in sync by incrementally forwarding the state changes as soon as they occur. Alternatively, it is possible to buffer a larger set of changes, and migrate them to other devices as a batch. For seamless real-time updates at the UI level, the trickle approach is mandatory. However, since many devices partaking in liquid software scenarios may be offline for prolonged periods of time, batch updates typically need to be supported as well, so that previously recorded changes can be "played back" on other devices as those devices become available online again. An obvious challenge in buffering changes and transmitting them later when connectivity is restored is that devices may be in inconsistent state and require reconciliation [28].

No matter which approach is chosen, a procedure that synchronizes the entire system is needed when initiating the execution of an application on new devices. Depending on the mechanism that is used for launching new applications, this can take place either using a central server or in a P2P fashion. In addition, conflict resolution between different devices requires a protocol for agreeing over the common state. Depending on the situation, this may again happen via a central

server or, e.g., by voting among the clients themselves. A simple but effective solution chosen by Koskimies *et al.* [108] was to allow the latest change to override any past conflicting changes in order to avoid any deadlocks or communication overhead associated with voting.

The choice of the state synchronization alternative may also impact the way how the developer controls the synchronization and how synchronized elements of the data are identified. While the migration [26] or the synchronization [41] of the state of an entire virtual machine can be done as a batch operation, the trickle approach can also work with finer-grained abstractions, such as applications or individual components. To do so the developer should have mechanisms to explicitly indicate which parts of the state should be moved to the new location.

Federation of synchronization. An important consideration in liquid software system development is the federation of devices that can partake in the migration of data and state. In multi-device scenarios it is important to be able to carefully manage access control rights and grant permissions depending on the ownership of the device on which the software dynamically finds itself running on. We identify two basic permissions controlling the direction of synchronization:

- *Publishing*: the ability to send/push data to paired devices.
- *Subscribing*: the ability to receive/pull data from paired devices.

These permissions are particularly useful in multi-user scenarios, to make sure both parties agree to exchange data.

3.2.8 Privacy and Security

The success of computing platforms supporting liquid behavior is fundamentally dependent on *security*. As summarized by Taivalsaari *et al.* [174], the ability of liquid software to readily flow from device to device is both a blessing and a curse. It is a blessing because it enables a new computing paradigm, virtualized but personal computing environment that is independent of any specific computer or device. However, the very mobility of liquid software is a curse because it can open potentially huge security holes. The notion of the user's entire computing environment being accessible from any of the user's devices can make the system vulnerable from a security and privacy perspective. For instance, if even one of the user's devices is stolen, there is a possibility that his entire computing environment could be compromised.

As a starting point for security and device federation, there are well-known techniques for secure communications, device pairing and trust establishment, user authentication and authorization that are needed for implementing secu-

rity features for any liquid application. These techniques have been maturing for years in the context of computer networks, the Web, cloud computing, and mobile devices. These already existing mechanisms can largely be used to satisfy the requirements for privacy, cohesion, authentication, authorization, and audit.

A basic principle defined by Taivalsaari *et al.* [174] is to keep the user in full control of the liquidity of applications and data. This calls for a security approach that is flexible yet simple and straightforward in layman terms, not assuming special skills or a deep understanding from the end user's part. For example, the SunRay ultra thin network terminals [Ora16] provided a secure smart card authentication system that would connect the client device to the remote user session, making it appear truly as if the user's earlier computing session had instantly migrated to the present target terminal. More work is needed to investigate which authentication techniques and security practices can be accepted by the end users in different usage scenarios.

3.3 Maturity

Liquid software does not directly depend only on the technology used to develop it, rather it depends on how its development is planned during the development process [136]. In order to understand how to create multi-device applications we must be able to understand the driving quality attributes [142; 11], specifications [165], risks [51], and requirements [183] of liquid software beforehand. By considering multi-device interactions and seamless migration between devices we are adding new degrees of complexity to the applications if we compare them with *solid* counterparts; therefore we must address new Web engineering challenges in order to design and eventually deploy applications that can spawn on multiple devices.

As multiple users can operate multiple devices at the same time, the architecture of the applications needs to be redesigned with new requirements in mind. The liquid software must take full advantage of the connected devices' hardware, computing power and communication resources. The primary quality drivers of liquid Web software are **compatibility**, e.g., **portability** on different platforms, and high **reliability** when running on cross-device environments [174]. Moreover the top most important quality attribute for the expected LUE is **usability**, specifically **learnability**.

Web applications were traditionally developed following a thin client architecture whereby most of the logic and the entire persistent state of the application would be executed and stored on a central Web server. They would offer only

partial support for the LUE in terms of the ability of migrating the application by simply sharing the URL pointing to the current state of the application and adapting the UI by employing responsive Web design techniques [125].

As the Web technology platform has evolved with enhanced support for rich and thick client architectures and for protocols beyond Hypertext Transfer Protocol (HTTP) to enable real-time push notifications and P2P (browser to browser) connections, we revisit the architectural design space of contemporary Web applications to systematically study new deployment configurations and how these impact the LUE.

We derive the maturity model of liquid Web applications to assess how different Web application architectures can provide support for the LUE for both sequential and parallel screening scenarios:

- **Sequential screening**: users own more than one device, at any time they may decide to continue their work using another device. The application and the associated state seamlessly flow from one device to another;
- **Parallel screening**: users own multiple devices and deploy the software on all of them. Users may decide to change the number of devices running the liquid application as well as to move components of the application from one device to another while keeping the state up to date.

Web developers can follow the maturity model to redesign, refactor and transform their applications to provide enhanced level of support for liquid behaviors defining the following LUE primitives:

- **Forward**: the ability of an application of redirecting the input/output of one device to another;
- **Migrate**: the ability of moving a running application to another device;
- **Fork**: the ability of creating a perfect copy of an existing application on another device.
- **Clone**: the ability of creating a perfect copy of an existing application on another device, and keep the state of the original and the copy synchronized thereafter.

Sequential screening can be achieved if an application defines either a migrate or fork primitive, while parallel screening can be supported either with clone or forwarding primitives.

3.3.1 Maturity Model Facets

The maturity model of liquid Web applications is based on multiple facets that determine the degree of *liquidity* of a Web application both in terms of which LUE primitives are enabled as well as how these can be implemented with different

performance and privacy guarantees. Each architectural configuration presents unique challenges and opportunities to deliver a liquid behavior under different constraints. For example, while it is relatively easy to synchronize the state of the application relying on a highly available, centralized master copy deployed in the Cloud, some privacy and latency issues may warrant considering more decentralized or distributed approaches to data management.

The maturity model is based on three orthogonal facets, each having three levels:

- logic deployment (ultra-thin, thin, thick);
- state storage (centralized, decentralized and distributed);
- communication channel (HTTP, WebSockets and WebRTC).

In this section we provide a systematic discussion on the implications of the most significant architectural configurations on whether and how liquid Web applications can be built under the corresponding architectural constraints. Additionally, for each level, we survey existing Web development frameworks within the corresponding Web application architectures. As we are going to show, migration of Web applications can be achieved with all configurations, while cloning requires support for real-time synchronization that is only present in higher maturity levels.

Web applications comply with the client-server architectural style, in which persistent resources or services are provided by one server to multiple clients. Without loss of generality, we further describe Web application architectures using the MVC pattern, one of the most used design patterns in Web applications development [94]. In the MVC pattern Web applications are logically decomposed to manage separate concerns: data modeling and persistent storage, data processing and business logic, and data input/output and user interaction.

- The **Model Layer** manages the persistent data of an application. The model of a Web application also includes any of its assets such as Web Pages, images, and scripts that need to be transferred to the clients. This layer requires some kind of data storage able to represent, organize and collect information:

– in the server-side of an application it usually takes the form of a database [DBE20] such as relational databases like *Oracle* [27] and *MySQL* [190], document oriented databases like *MongoDB* [27], or *Apache CouchDB* [7], or other schema-less databases like *Redis* [31];

– in the client-side usually the file system of the device is used as storage, but due to the possibility of having clients running on heterogeneous devices and implemented using different programming languages, data storages can highly differ from client to client even in the same application. *WebSQL* [W3C10], *IndexedDB* [Moz19c], *LocalStorage* [Moz19d; Moz19e], and *Cookies* [15] are stan-

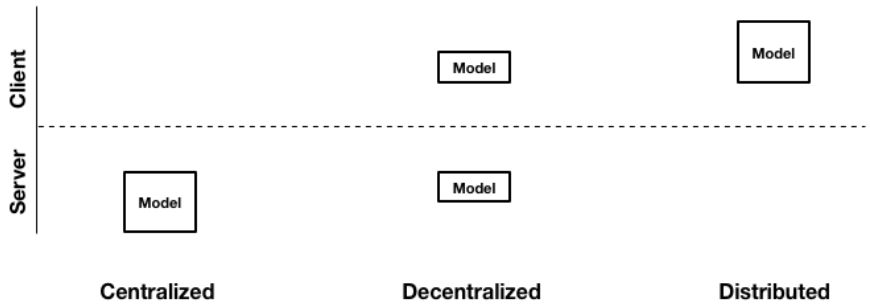


Figure 3.8. Model layer deployment levels

dard implementations of data storage APIs available in HTML5-compliant Web browsers.

• The **Controller Layer** consists of the logic of an application. The controller layer is a bridge between the model and view layers, it manipulates data and executes tasks received from either layer and forwards the results to the appropriate one. Depending on where the controller layer is deployed it can be implemented using different programming languages. In the server-side *PHP* [190], *ASP.NET* [Mic20], and *JavaScript* (using *Node.js* [177]) are the most used programming languages, while in the client-side *JavaScript* is the main option.

• The **View Layer** is the graphical UI of an application, consisting of the visual representation of the data and information retrieved from the model layer and rendered into an interactive visualization.

Combining the client/server execution environment and the three MVC layers, we identify different deployment combinations. While the view layer is constrained to run on the client, both model and controller Layer can be deployed on either side (or partitioned to run on both client and server). Additionally, we distinguish three alternative communication channels and protocols (HTTP, WebSockets and WebRTC) used to interconnect the layers of Web applications running on different devices.

In the following sections we discuss more in detail each facet which will be combined into the liquid Web application maturity model in Section 3.3.4.

3.3.1.1 Model Layer deployment

Model layer deployment describes where the persistent state of the Web application is stored. We identify three levels based on whether data is logically centralized on the server or distributed towards the clients (see Figure 3.8):

Level 1 - Centralized - The model is stored using any data management solution that is solely deployed in the server-side. For scalability and availability purposes, the actual storage can be implemented using multiple virtual servers running in a Cloud data center. Conceptually, this is still a centralized solution as data is never managed by the client. The advantage is that no matter what client device is used to access it, the data will be readily available [167]. Users thus trade off the convenience of accessing "their" data anywhere with the loss of control over the actual location where it is stored and who else can access it. As clients always need to remotely request data from the server, there are also latency and availability implications to be considered. When multiple clients perform transactions to update shared resources, having a single master copy on the server helps to ensure consistency.
Some real world examples of centralized model layer deployments use databases created with *MySQL*, *MySQL Cluster*, or *Cassandra* [119].

Level 2 - Decentralized - The model layer is deployed both in the server and client-side of the Web application. Information stored in the server database is replicated or cached by the clients. Conversely, users may prefer to save the primary copy of their data in their own clients and use the server as a secondary backup.
Cookies are the simplest example of decentralized persistent storage on the Web. Web application using any technology mentioned in level 1 (e.g. *MySQL*) in combination with any HTML5 storage API (e.g. *localstorage, WebSQL*) falls in this category. *Apache CouchDB* or *PouchDB* [Pou20] are databases that feature client-side caching with automatic synchronization allowing offline availability of the retrieved data.
Decentralized approaches enhance: **–** data privacy, even though data must still be transmitted to the servers if there is not a direct communication channel between clients; **–** availability during offline operation, assuming the data has been prefetched by the client the Web application may still work while being disconnected from the server; **–** enhanced perceived performance when hitting data cached on the client.

Level 3 - Distributed - The model layer is distributed exclusively on the client-side of the Web application. There is no need to use the server to retrieve or store data, because clients completely own the state of the Web application. This positively impacts data privacy because the information of the users always remains on their devices and is never stored in a Web server outside of their control (e.g., in the Cloud).
Distributed model layer deployment can be achieved in a modern Web browser by using any combination of the storage APIs provided by the HTML5 standard,

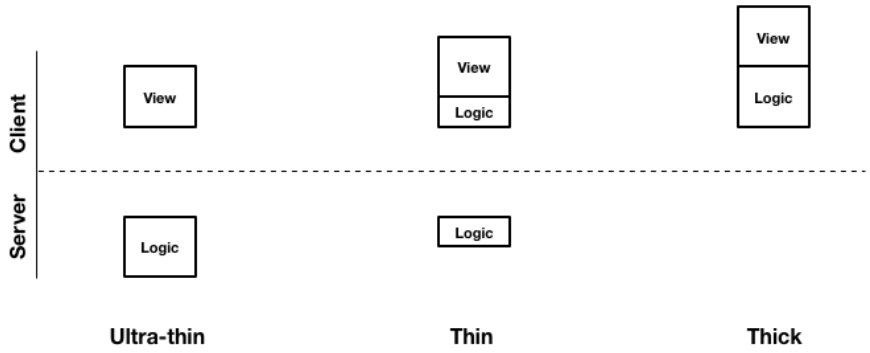


Figure 3.9. Controller layer deployment levels (labeled as logic)

namely the *WebSQL*, *IndexedDB*, and *LocalStorage* APIs. On top of these technologies, or even using the file system of the devices running the client of the Web application it is possible to build distributed model layers able to automatically synchronize between clients (e.g., [188]).

3.3.2 Controller Layer deployment

The Controller layer deployment determines where the Web application executes tasks and whether it can offload its workload. We define three levels with respect to the client thickness (see Figure 3.9):

Level 1 - Ultra-Thin Client - In this level the controller layer of an application is deployed only on the server-side of the application. The only logic present on the client is the logic needed to retrieve content from the server and to display views when they are received from the server.
Primitive Web browsers that did not allow running scripts, such as *JavaScript* or *Java Applets* can be seen as *ultra-thin* clients. Ultra thin clients always display the view layer statically and cannot adapt it to the client's device. *Curling* pages on a terminal is also an example of ultra-thin Client, in which the forwarded raw data is displayed. Web applications that do not require scripts to run in the client fall in this category as well.

Level 2 - Thin Client - The logic of an application is deployed on both server and client-side of the Web application. The server can offload part of the computations to the connected clients. The most offloaded task in Web applications is the creation of the views which is entrusted directly to the clients needing it, however in thin clients any simple task can be offloaded to the clients.
Whenever the client is thin, it is possible to make views responsive to the client's

device. This allows the same applications to use a different look and feel in devices with different hardware specifications.
AngularJS [Goo18], *React* [Fac20], or *EmberJS* [Til20] are some frameworks for isomorphic Web applications written in JavaScript that require thin clients.

Level 3 - Thick Client - The logic of an application is entirely deployed on the client-side of the Web application. A big portion of the application computations are offloaded to the clients. As in level 2 clients compute the views they display. Additionally they execute computationally-heavy application-specific tasks that were not previously included in thin clients. The HTML5 WebWorker specification allows Web browser to run scripts in background, making it possible to develop complex client-side applications [33].
Thick clients can be aware of other connected clients, making it possible to adapt the view layer of the application on a set of devices instead of making it responsive to a single one. Complementary adaptive views can also automatically evolve in real-time if the application is able to propagate to all devices the knowledge of connections and disconnections of other clients.
Web 2.0 single-page applications generally require thick clients, any client described in level 2 can become a thick client if the entire controller logic layer is deployed in the client-side. Liquid Web applications featuring all the LUE primitives require clients to be thick.

3.3.3 Communication channel

The communication channel facet is characterised by the direction of the communication between the client and the server and whether clients can communicate directly. The levels shown in Figure 3.10 are inclusive, whereby a higher level also includes all the features provided by the lower levels:

Level 1 - Client-Server Pull - Clients are always the origin of all request-response interactions with the server. Clients request resources addressed by Uniform Resource Identifiers (URIs) and the server responds with the corresponding representations if they exist. On the Web, this kind of communication is implemented with the HTTP protocol.
Applications relying solely on the HTTP protocol cannot propagate state changes or events occurring on the server back to the clients in real-time. They can only simulate a quasi-real-time environment (with continuous polling). While the liquid migrate primitive can be implemented with HTTP only, cloning or forwarding cannot be implemented in level 1, because data synchronization in liquid applications requires real-time notifications.

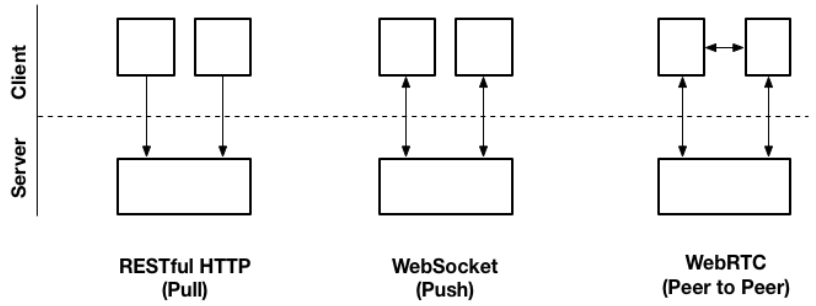


Figure 3.10. Communication channels

Level 2 - Client-Server Push - Similarly to the client-server pull level, clients are still the origin of the interaction with the server. However in level 2, clients open a two-way communication channel. In this level the server is allowed to propagate data and events to the connected clients immediately, meaning that it is possible to efficiently create real-time Web applications. The standard Web protocol used for implementing client-server push is WebSocket. With WebSocket it is possible to implement the liquid clone primitive since data synchronization can happen in real-time. Liquid Web applications whose goal is to implement all possible LUE primitives need to consider at least a level 2 communication channel in the design of their architectures.

Level 3 - Peer-to-Peer - With the advent of the WebRTC protocol it is now possible to have P2P communication among Web browsers. Architectures implementing level 3 communication channels still rely on the HTTP and WebSocket protocols for peer discovery purposes. Level 3 communication channels allow to lower the latency between clients, by potentially decreasing the number of hops in the communication, instead of propagating data relying on the server (client → server → client), in the best case it is possible to communicate directly between clients (client → client).

3.3.4 Maturity Model

Figure 3.11 shows the maturity model of liquid Web applications determined by combining the deployment configuration of their MVC layers across the server-side and client-side with the choice of the communication channels established between them. We identify five levels: 1. Web 1.0 Applications 2. Rich Web Applications 3. Real-time Web Applications 4. Hybrid Web Applications 5. Peer-

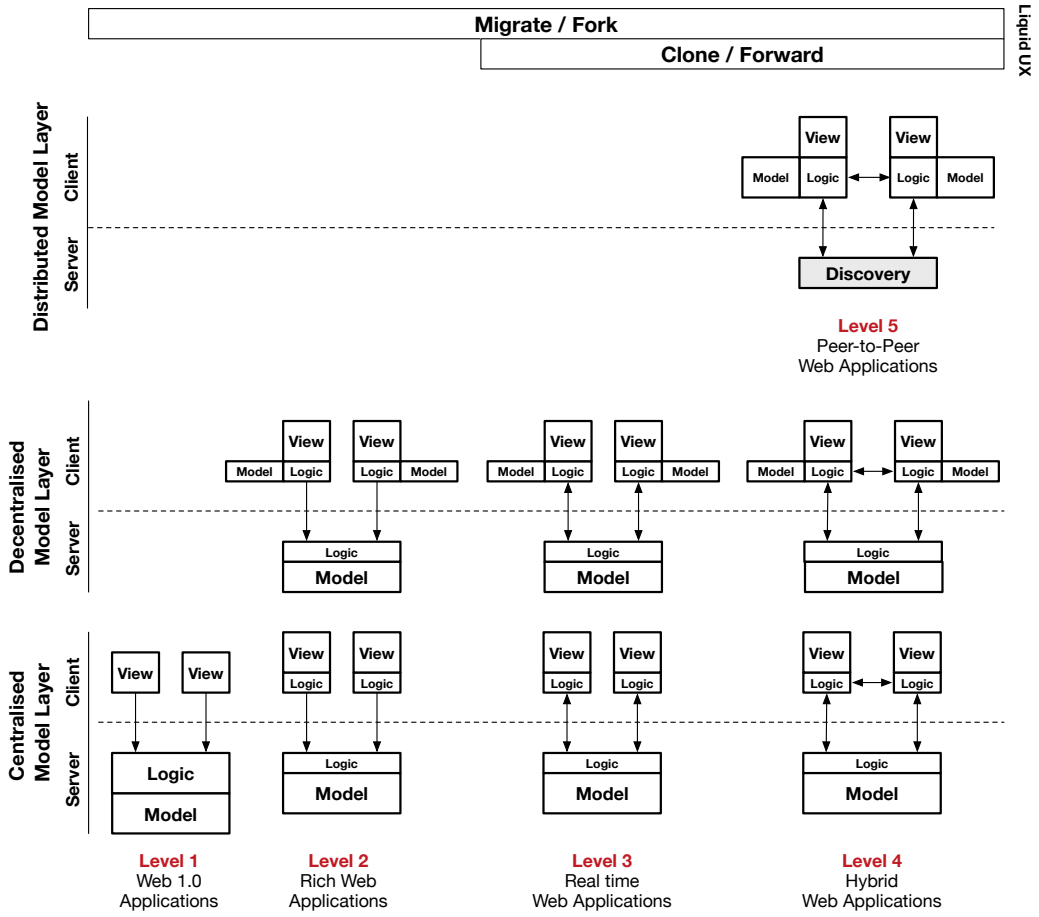


Figure 3.11. Maturity model for Web architectures for centralized, decentralized and distributed model layer deployments. The controller layer is labeled as Logic.

to-peer Web Applications

The diagram also shows in which level is possible to use the *migrate/fork* and *clone/forward* LUE primitives. Migrate and fork are possible in all levels, while clone and forward can only be achieved starting from level three.

Table 3.2 summarizes the different configurations per level as a combination of the facets explained in Section 3.3.1. The table also describes how the configurations affect the following quality attributes:

- **Latency** or the proximity between two clients on the network:

– *2* hops means that whenever clients communicate to perform a LUE primitive, the communication relies on an intermediary Web server (client → server → client);

Table 3.2. Maturity model: architectural configurations and quality attributes.

	Configuration			Quality Attributes				
	Deployment		Channel	Latency	LUE primitives		View	Privacy
Level	Logic	Model		Hops	Migrate Fork	Clone Forward	Adaptation	Model
1	Centralized	Ultra-Thin	Pull	2	A	✗	Static	✗
2c	Centralized	Ultra-Thin	Pull	2	A	✗	Responsive	✗
2d	Decentralized	Thin	Pull	2	A	✗	Responsive	✗
3c	Centralized	Ultra-Thin	Push	2	A, S	✓	Complementary	✗
3d	Decentralized	Thin	Push	2	A, S	✓	Complementary	✗
4c	Centralized	Ultra-Thin	P2P	1 or 2	A, S	✓	Complementary	✗
4d	Decentralized	Thin	P2P	1 or 2	A, S	✓	Complementary	✓*
5	Distributed	Thick	P2P	1	S	✓	Complementary	✓

– *1* hop means that clients – in the best case – can communicate directly with each other.

• **LUE primitives**:

– *Migrate/Fork* can occur asynchronously (*A*) or synchronously (*S*). *Asynchronous* migration and fork of an application happens between two clients that cannot directly push the migrated state and logic to the target client, but they have to be stored in a central storage first; *Synchronous* migration and fork of an application can be implemented in systems in which clients can push migrated or forked state and logic to other clients without the need to store it in a central storage.

– *Clone/Forward* indicates in which configurations the liquid clone and forward operations are possible ✓or not possible ✗.

• **View Adaptation** describes which level of the view layer adaptability is possible to achieve in all configurations:

– *Static* means that the view does not dynamically adapt to the client hardware device capabilities, but it is displayed exactly as it was determined by the server;

– *Responsive* means that the view locally adapts to the client.

– *Complementary* means that it is possible to manually or automatically adapt the view to *set of* heterogeneous devices connected to the Web application.

• **Privacy** describes if the users have control on their data by ensuring that it is exclusively stored in devices they own or trust: ✓ means that users are in control of their own data, ✗ means that the data is stored in untrusted devices, ✓* means that the data is stored in trusted devices, but is exchanged with or relayed across untrusted devices (e.g., a Web server running in the Cloud).

3.3.4.1 Level 0 - Solid Applications

All layers of a solid (or monolithic) application are deployed on the same machine, typically a standalone personal computing device, or a server to which multiple dumb terminal devices are attached. This architecture configuration predates the Web as intra-layer communication does not go through any Web protocol, but only happens locally within the same host using, e.g., local procedure calls or shared memory buffers.

Liquid migration can be achieved by the mean of input/output virtualization of the clients, e.g., multiple users can access the operating system installed in the server by different screens. This architecture allows users to save their data on the server and access it from any screen. From the user perspective, switching terminal device amounts to successfully migrating their work from one screen to another. The *virtualized* client is therefore ultra-thin, where the view layer running on the server forwards the UI input/output events and commands to the terminals connected to it.

The concept of the first Sun Ray [Ora16] designed in 1997 can be considered as an example level 0, solid application architecture. The concept is designed to take advantage of stateless network computers whereby users authenticating with smart cards were instantaneously taken to their virtual desktops and could access their applications and data centrally managed on the server from anywhere.

3.3.4.2 Level 1 - Web 1.0 Applications

Level 1 Applications can be seen as the first generation of Web applications [17] built using the HTTP protocol. The logic and model layer are deployed on the server-side, while the view layers run client-side on Web browsers. In this level the content provided by the Web servers is static and cannot be changed by the clients. Web browsers retrieve content (e.g., Web pages written in Hypertext Markup Language (HTML)) by sending HTTP requests to Web servers addressed by URIs. Browsers display resources as they were sent from the server. The view layer is completely static, since there is no definition of a technology able to adapt the retrieved resources to different client rendering capabilities. At that time Cascading Style Sheets (CSS) media queries did not yet exist while the Extensible Stylesheet Language (XSL) did not provide any markup to adapt the content of a Web page to the device displaying it.

Level 1 supports asynchronous liquid migration by uploading resources to the server and then using their URL to retrieve the resource from another device.

In this level cloning a liquid Web application is challenging to achieve, because data synchronization does not happen in real-time, as clients can only resort to continuous HTTP polling. Likewise, migration in level 1 does not happen in real-time and requires the exchange or agreement on the URL addressing any resource that needs to be migrated between clients, which do not need to be available and connected at the same time.

This is the most basic architecture for implementing liquid applications that only need the liquid migration primitive, it does not provide any kind of view adaptation and cannot ensure data privacy (unless the Web server is owned and operated by the same organization owning the client devices). Synchronization between multiple clients can be achieved only by manually refreshing the Web page after sharing URL via out-of-band channels, which does not fit with the real-time expectations of the LUE.

3.3.4.3 Level 2 - Rich Web Applications

In Level 2 we consider rich Web applications [33] in which the controller layer is deployed both in the server and client-side. Level 2 architectures are the first ones able to have responsive views because the portion of the controller on the client-side can compute different views and do so based on the underlying hardware capabilities. Liquid migration is possible, but, like in level 1, shared URLs are needed to address and retrieve the resources representing the state to be shared among clients, since there is not a direct communication channel between clients. More in detail, after discovering the identifier of the resource being migrated, the Web application on the browser is manually refreshed to ensure the consistency of the displayed information with the model of the Web application. Again, cloning is hindered by the lack of real-time communication between clients attempting to immediately synchronize. The distance between two clients is always equal to two hops.

Depending on how the model layer is deployed on the clients we distinguish two different level 2 configurations: Level 2c - **centralized -** the model is deployed only in the server-side; Level 2d - **decentralized -** part of the model is stored in the client-side, in traditional Web applications it takes the form of *Cookies* or cached data, in modern rich Web application it takes various forms (e.g., local storage, service workers, WebSQL databases) as described in Section 3.3.1. Users of a liquid rich Web application can store their confidential data in their devices, nevertheless during a migration of the liquid Web application its state has to be transferred via the server, which may not be always owned or trusted by the user of the Web application.

The LUE in this level is similar to the one in level 1, with the addition of support for a responsive view and the option of storing parts of the model locally on the client.

3.3.4.4 Example Level 2 Frameworks

CrowdAdapt [141] is a centralized level 2 framework for creating responsive Web pages. Web pages created with CrowdAdapt allow users to change the layout of the Web page as they desire and thereafter migrate their creations on other devices. In CrowdAdapt the controller layer provides the editing functionality and the automatic detection of the hardware specification of the device running the client. The users are able to choose between the layouts created by all the users of the Web application that better fit their needs.

PageTailor [19] is a decentralized level 2 framework with concepts similar to CrowdAdapt. Users can change the layout of Web pages using PageTailor and then reuse these layouts on subsequent visits. In this case layouts are not shared between multiple users as in CrowdAdapt.

3.3.4.5 Level 3 - Real-time Web Applications

The deployment of the view and controller layers are the same as in level 2, however level 3 applications have access to client-server push communication channels. This make it possible for Web applications to synchronize data among clients and notify connections of new clients and detect disconnections of old devices. The liquid clone operation is implementable in level 3 because data can be synchronized in real-time between simultaneously connected devices. The awareness of the connected clients to the Web application allows to distribute the view layer among them. The complementary view implementable in level 3 Web applications increases the quality of the LUE. Liquid migration, liquid cloning and complementary view control can happen at different granularity levels: **application level -** the Web application is monolithic and all devices receive all the assets and model of the whole application. Upon migration or cloning the new clients have a perfect copy of the whole application whose state is kept synchronized between them. Complementary view adaptation in this case can be implemented through *Web clipping* by concealing part of a view on all but one device. **component level -** in component-based Web applications clients receive only portions of the whole application. Liquid migration and cloning can be done at component level, thus moving and keeping synchronized only part of the application. In this granularity level complementary view development

does not need *clipping*, because clients move or receive only the portions of the application they need and do not have to locally hide the components displayed on other devices.

The **decentralized** configuration of level 3 allows partial privacy on the data created by the users, as it can be stored only on trusted devices. During liquid migration or cloning on simultaneously connected devices there is no need to store any information on the server. However data sent between clients is still relayed through the WebSocket channels on the server, meaning that such data must be encrypted in order to ensure privacy. In the case of liquid migration on devices which are not simultaneously connected to the Web application, the entire model has to pass through the Web server regardless.

3.3.4.6 Example Level 3 Frameworks

Smart Composition [113] is a centralized level 3 framework that allow the creation of component-based (called widgets) multi-screen Web applications. By using a central *cross-device communication service* the infrastructure created by SmartComposition is able to compose distributed view layers among devices and keep the various components building the application synchronized.

Panelrama [192] is a centralized level 3 framework used to create distributed UIs using the concept of *panels*, JavaScript objects defining pieces of UI and logic. Panelrama provides an API to migrate and clone panels between devices and automatically create the complementary distribution of the view layer among the connected devices.

DireWolf [109] is a decentralized level 3 framework used to create multi-device mashups Web applications. Clients are aware of the connected devices in the application and can migrate widget-like components to any target device. DireWolf offers the possibility to manage the device ownership, device information and specification, the widget state, and the application state of the whole application through its clients.

Liquid.js for DOM [179] is a decentralized level 3 framework based on *React.js* for component-based Web applications. By synchronizing virtual Document Object Models (DOMs) between devices it is able to migrate and clone logic and model layers among the connected clients. It does not offer automatic cross-device complementary views. There also exists a level 4 Hybrid version of Liquid.js for DOM offering peer-to-peer data synchronization between clients.

Bellucci *et al.* [14], Frosini *et al.* [54], and Raposo *et al.* [153] propose similar frameworks in which is possible to distribute and synchronize the view layer of the application on all connected devices.

3.3.4.7 Level 4 - Hybrid Web Application

Level 4 augments level 3 with the ability for clients to communicate directly with each other through P2P channels. The logic layer deployed in the client-side can send messages to other clients either directly with a single *hop* or through the server with two *hops*. Connected clients can send any kind of data between each other, including the entire assets of the application. Similarly to level 3, it is possible to have both asynchronous and synchronous migration and cloning operations among connected clients. Through the P2P channels decentralized hybrid Web applications can send confidential data directly among trusted clients without relaying any message through the server, ensuring privacy if confidential data does not need to be stored on the server.

3.3.4.8 Example Level 4 Frameworks

XD-MVC [89] is a decentralized level 4 framework for creating cross-device interfaces and automatic complementary adaptation views applications. XD-MVC implements migration at the application level and takes advantage of clipping off parts of the view layer in order to simulate migration between devices. Views can be annotated with rules about how they are expected to adapt to the set of connected devices. Given these rules the view is able to dynamically and automatically adapt to set of heterogeneous devices when a new client connects or disconnects.

PolyChrome [13] is a centralized level 4 framework for creating co-browsing applications with collaborative views spanning on multiple screens deployed on multiple devices. PolyChrome complementary view adaptation supports *stitching*, *replication*, *nesting*, and *overloading* layouts. Data synchronization happens both through P2P and WebSocket channels. The framework creates components out of a legacy applications in order to be able to make a view span on multiple devices.

3.3.4.9 Level 5 - Peer-to-peer Web Applications

Peer-to-peer [163] Web architectures are at the highest level of the maturity model, the only one providing all quality attributes expected from liquid Web applications. P2P Web applications allows connected clients to communicate with each other directly. When peers are linked with a fully-connected mesh, this amounts to the best case scenario with a latency of 1 hop. Indeed other topologies are possible, like rings, in which N connected clients are up to $N/2$

hops away, or stars, in which the hops number vary between 1 and 2, depending on which peers are communicating.

Level 5 applications allow synchronous migration, since connected clients can push the model and logic through the full-duplex P2P channel created with WebRTC at any time. Since there is no longer a central server available at all times asynchronous migration is not possible. Instead clients need to be online simultaneously in order to proceed with the migration. Since clients sense and propagate their availability across the peer to peer network, it is possible to have complementary view adaptation.

Data privacy is ensured in P2P Web applications because users are in full control of all devices storing and processing their data. Data is never stored in any server or Cloud storage platform. Also, data migrated or zed with another device is never sent through a Web server.

Level 5 Web applications allow strong mobility with direct model and logic transfer and synchronization between peers, however this requires to a suitable discovery method. WebRTC, for example, uses a signaling server to initiate and establish the connection between clients. Clients first connect to the signaling server and then receive information on how they can join the rest of the peers. Once the topology is created and peers are connected, they are free to communicate among themselves and the signaling server is no longer involved.

3.3.4.10 Example Level 5 Frameworks

Liquid.js for Polymer [66] is a level 5 P2P framework which allows the creation of distributed component-based Web application built on top of the Polymer framework. Users instantiate any component provided by the Web application on their devices and share them directly with other users. If a peer does not own the assets of the component being sent to it, the peer will also receive the model of the component that is going to be migrated. Liquid.js allows to define strategies for creating different peer topologies.

3.3.5 Beyond Level 5 Framework

Each level's architectural configuration impacts the possible LUE primitives. Most of the existing liquid Web application development frameworks [13; 14; 19; 54; 66; 89; 109; 113; 141; 153; 179; 192; Ora16] we surveyed are categorised by a level 3 (real-time) architecture, however we emphasize that higher level architectures are possible and they should be considered to deliver all the quality attributes that one would expect from a liquid Web application, in particular

data privacy (with decentralized configurations) and a reduced latency between devices that do not need to communicate with or through a remote Web server all the time. We acknowledge that real-world liquid applications with multiple client implementations may span multiple levels in the maturity model. For the sake of simplicity and clarity we described the main five levels in the maturity model instead of all possible combinations.

The choice of level and configuration should be implemented or which framework to use are important architectural decisions. Upgrading an architecture from a lower level to a higher one, or downgrading to lower levels fundamentally impact the design of the Web application and are are likely to result in significant development costs. Still, over the history of the Web, application architectures have been gradually and steadily shifting towards the higher levels of the maturity model.

As the number of devices connected to the Web and the average number of devices owned by one user increases [Glo17], more frameworks will appear positioned across all levels of the maturity model targeting the creation of liquid Web application. An evaluation of the presented and future frameworks in terms of performance, scalability, and usability would allow developers to assess which framework is more suitable for executing liquid primitives in the sequential and parallel scenarios.

HTML5 standards are quickly evolving every year and new specification drafts are already defining new technologies that may be used to further extend and improve the LUE provided by liquid applications. While we described five levels in our maturity model, we do not exclude that in the future higher levels will appear. For example, emerging technologies like Web Bluetooth (currently not yet a World Wide Web Consortium (W3C) standard) [Web17] aims to bring Bluetooth support in Web browsers which may be used to define a new maturity level in which there is no longer a need for a central server in order to perform client discovery and device pairing.

We based our description on the MVC design pattern adopted by traditional Web applications, however in the future it may become necessary to revisit the fundamental architectural abstraction and design principles of Web applications and study their interplay with a programmable world [172] of billions of heterogeneous interconnected devices.

Part II

Liquid Web Architectures

Chapter 4

Liquid Data Layer and State Synchronization

The design of the data layer and communication channels of a liquid application is essential for creating liquid Web applications featuring seamless migration across multiple devices, because the applications' data and state can flow only if the underlying data layer allows the applications to migrate across the devices. For this reason we start this part by discussing the design of the liquid data layer.

In this chapter we focus our discussion on how to design the data layer of liquid Web applications positioned in the level 4 and 5 of the maturity model presented in Section 3.3. The concepts presented in this chapter can also be used for lower levels in the maturity model, but their design should be revisited in order to accommodate the deployment of the other MVC layers.

4.1 Communication Channels

As we discussed in Section 3.3.3, the choice of the channels used in level 4 and 5 liquid Web applications is limited if we consider only the standard protocols accessible in HTML5 compliant Web browsers. In Figure 4.1 we show the most simple deployment of a level 5 liquid Web application:

• the clients can communicate with a signaling server through a WebSocket channel, or through continuous long-polling HTTP requests. In the Web browsers these HTTP requests can be natively implemented by creating multiple AJAX requests that can be chained by using the *XMLHttpRequest* API [Moz19f], or by chaining requests created with the newer HTML5 Fetch API [Moz19g]: with the former, the developers must design the chain of requests with callbacks; with the latter, developers can exploit the Promises paradigm [Moz20c].

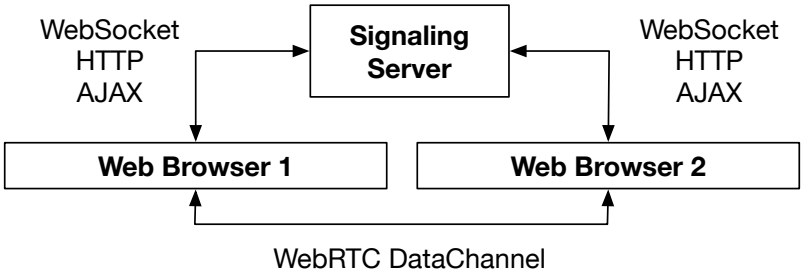


Figure 4.1. Deployment view of a level 4 or 5 liquid Web application.

The top two most popular and downloaded WebSockets libraries, that can be used for signaling servers implemented on top of a *Node.js* process [Ope20], are *ws* [WS 20] and *socket.io* [Soc20]: – *ws* is a lightweight WebSockets protocol implementation for the server-side that can create fast client-server communication channels. The library does not automatically provide any fallback in case the standard WebSockets are not accessible or available on the Web browser. Developers must therefore take care of the fallback implementation themselves. – *Socket.io* is a collection of protocols that guarantees the two-way communication between the clients and the server. The library must be loaded both in the client and in the server, and can transparently switch between communication protocols if the standard WebSockets are missing or unavailable in the Web browser.

• clients can communicate with each other by using the WebRTC DataChannels which can be created through the *RTCDataChannel* and the *RTCPeerConnection* interfaces in any modern Web browsers [Moz20a; Moz19h]. It is important to note that WebRTC can create different communication channels suited for streaming different kinds of data, e.g., binary, audio, video, or text; however, for applications that do not need to stream audio, nor videos, the DataChannels should be enough to synchronize the data and state across multiple Web browsers. WebRTC is currently the only native implementation of a P2P communication channel protocol that can be used across Web browsers. At the time this dissertation is written, the majority of the modern Web browsers have access to the WebRTC DataChannels [Ale20], however not all implementations have access to every feature described in the WebRTC specification. The developers of a pure level 5 liquid Web application need to check if the features they plan to use in their applications are available on all Web browsers. There are many client-side libraries that help with the creation of WebRTC channels, however only PeerJS [Mic19] is focused solely on DataChannels, all other libraries focus

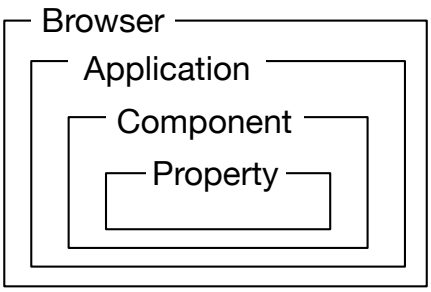


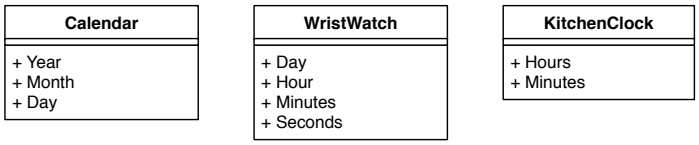
Figure 4.2. Liquid Web application granularity

on the creation of audio and video streaming channels, but do not always allow to create lightweight DataChannels. In case WebRTC channels are unavailable or the users do not give permission to access them on the Web browsers, developers must implement a fallback mechanism and relay messages through a Web server. The *Socket.io P2P* library [Tom20], similarly to *Socket.io*, is a collection of protocols that can create WebRTC PeerConnections between Web browsers that can also transparently switch to relaying messages through a Web server if the WebRTC channels fail to be created. The WebTorrent [Fer20] protocol, built on top of the WebRTC channels, can be used to speedup the synchronization performance of big chunks of data if it is distributed among multiple Web browsers.

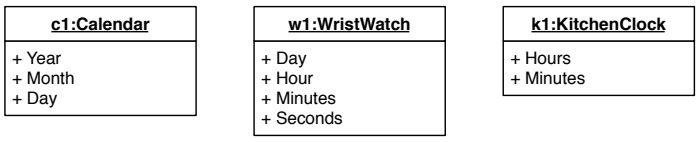
The data and state of a liquid Web application can also be synchronized across multiple Web browsers by using non-standard protocols. InterPlanetary File System (IPFS) [Pro20] can be used to share data between Web browsers by lowering the interoperability with clients that do not have access to IPFS.

4.2 Granularity

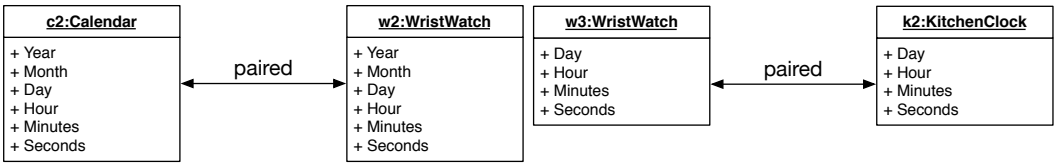
The granularity of a liquid Web application defines which parts of the liquid software migrates across devices. In Figure 4.2 we show the possible granularity levels that developers can consider when they design liquid Web applications using modern Web browsers. The more coarse-grained the liquid state is, the more data needs to be synchronized across multiple devices. Developers that design their liquid software with a Web browser granularity, need to address all the issues concerning the migration of the whole Web browser environment (e.g, a tab) from a device to another. The liquid software has to be able to spawn (or simulate the spawning of) a new Web browser process in the target machine and then populate and synchronize its data and state with a copy image of the source Web browser tab. Fine-grained liquid states, e.g., at the property level, synchro-



(a) Constructors



(b) Instances



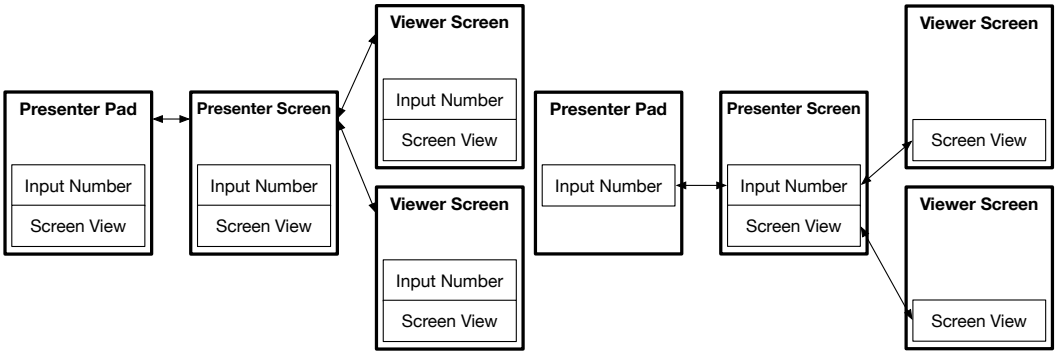
(c) Synchronized state between instances of Calendar and WristWatch.

(d) Synchronized state between instances of WristWatch and KitchenClock

Figure 4.3. Paired components state in component level granularity.

nize only smaller portions of an application across multiple devices, without the need of being aware of the underlying Web browser process state and thus without the need of synchronizing the whole application.

Coarse-grained liquid state layers (e.g., Web browser and application levels) do not need to explicitly specify which parts of the application they migrate Section 3.2.7, with the disadvantage that they synchronize more state across devices. Fine-grained liquid data layers (e.g., component and property levels) decrease the size of the data exchanged across devices, but with the drawback that the developers and/or the user need to annotate which components or properties are part of the liquid synchronized state. The property level is the pinnacle of the fine-grained liquid state, in which two components can share and synchronize the state of a single property and take advantage of it in different manners (e.g., two components that have the same logic but with two different visualizations). The more fine-grained the liquid state becomes, the more annotations are needed in order to further define it (see next Section 4.3) and the less data is exchanged between clients. The less data the clients exchange, the more responsive the liquid software becomes by increasing the seamless execution of the LUE primitives.



(a) Component level granularity (b) Property level granularity

Figure 4.4. Granularity example. Components are in bold and properties are stored inside components.

At the component level it is important to note that whenever multiple components are paired, the resulting synchronized state is equivalent to the union of the two (or more) separated component states. In Figure 4.3 we show an example of component instances created with different component constructors (Figure 4.3a). The state of the unpaired instances are equivalent to the state defined in the constructor (Figure 4.3b), however when the components are paired the resulting state is union set of the two (Figure 4.3 and 4.3d). While this effect is true for all granularity levels, it is impossible to notice it at the property level because a property holds the smallest indivisible piece of state.

In Figure 4.4 we show an example on how the design of the liquid data layer of a liquid Web application composed by three components can change with different granularity levels: • component level granularity is shown in Figure 4.4a; • property level granularity is shown in Figure 4.4b. The liquid application is meant to be used by a teacher who owns two devices (a smartphone and a computer) and from a class of students with their own laptop. The three components are: • *Presenter pad* component - meant to be deployed on the teacher's smartphone. The teacher can write any number on the presenter pad component for selecting the number of displayed slide. The number of the slide is stored in the *input number* property.; • *Presenter screen* component - meant to be deployed on the teacher's computer. The presenter screen component contains the properties *input number* and *screen view*. The component displays the current selected slide depending on the value stored in *input number* and save a screenshot of the displayed slide in *screen view*; • *Viewer screen* component - meant to be deployed on all students laptops. The viewer screen component displays the slide stored in the *screen view* property on all laptops. In Figure 4.4a the components are paired

as a whole, meaning that the data and state locally stored in any component is the same on every component. If we analyze the expected requirements of the components, we notice that the *presenter pad* component does not need to store the value of the *screen view* property because it is never used. The same can be said about the *input number* property in the *view screen* component, which is never used. In the more fine-grained example shown in Figure 4.4b we see that the liquid data layer is paired at the property level. The components do not need to store and synchronize all the state contained in all other components, thus they store only the values they need to fulfill the requirements. The *presenter pad* component only stores the value of the *input number* property and the *viewe screen* component stores the value of the *screen view* property.

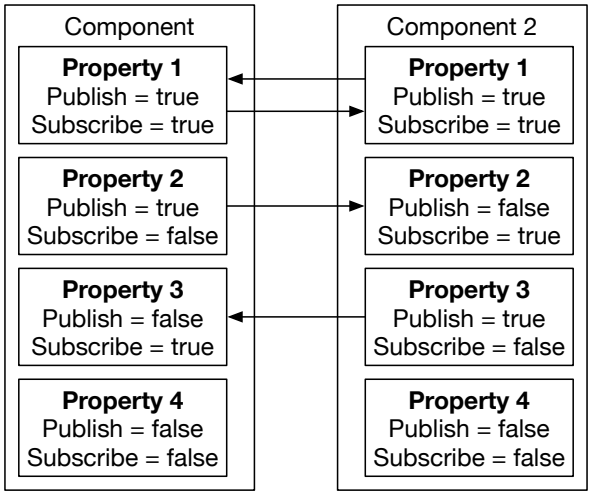
4.3 Data Flow Direction

In the example we presented in Section 4.2 we showed that the granularity of the liquid data layer effects the design of a liquid application, however the explicit state identification must also address other issues that can be explained using the example we presented in Figure 4.4. In the example the teachers expect to be in control of the application, and they do not want the students to change the slides they are displaying on the screen and on their computer. If a student can find a way to change the value of the *screen view* property, the value of the property would be automatically synchronized on the presenter screen, and then broadcasted to all students' laptop. While this could be a feature in an interactive class [175], it is a security breach not in this specific example. In this example the data is expected to flow from the *presenter pad* of the teacher, to the *presenter screen*, and finally arrive at the *viewer screen* of the students. The opposite flow of data is not allowed.

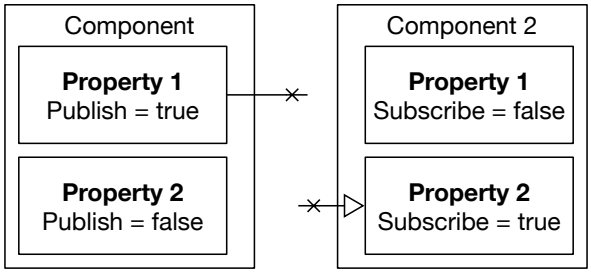
We introduce the *data flow direction* annotations for the explicit liquid data layer that can be used both at component and property levels of an application. The annotations consist of two permissions holding Boolean values which control how the data and state can be changed, updated, and propagated across paired components and properties. The permissions are:

- **publish** - defines if a component or a property is allowed to propagate its own updated state to another paired component or property;
- **subscribe** - defines if a component or a property is allowed to accept updates from another paired component or property.

The following discussion presents the publish and subscribe annotations at the property level, however the same concepts can be applied at the component



(a) Liquid properties with matching annotated permissions



(b) Liquid properties with mismatching annotated permissions

Figure 4.5. Data flow direction combinations for liquid properties.

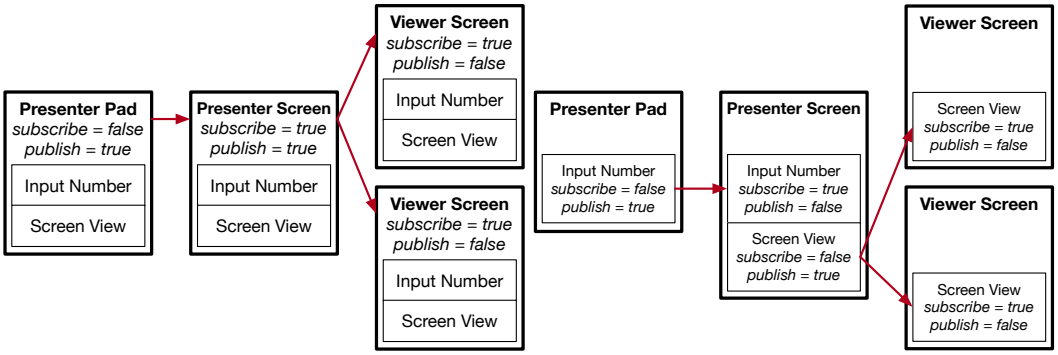
level.

Figure 4.5a shows the four matching combinations of the two permissions:

• *Property 1*: both *publish* and *subscribe* annotations are true in both components, thus *property 1* can flow in both directions. Whenever the property on the left is updated, the state is propagated to the property on the right and vice-versa;

• *Property 2*: the component on the left publishes property 2 and the component on the right allows property 2 to subscribe to other properties. In this scenario the data can flow only from the property defined in the first component to the property of the second. If property 2 inside component 2 is updated from outside (e.g., not from the transparent synchronization), is not propagated to the property defined inside the first component and is automatically re-synchronized to the previous value held by the the property on the left.

• *Property 3*: similarly to the example shown in property 2, the data can flow



(a) Component level granularity (b) Property level granularity

Figure 4.6. Data flow direction annotations example. Annotations are in italics, components are in bold and properties are stored inside components. This is example is the same as the one shown in Figure 4.4 but extended with the data flow direction annotations.

only in the opposite direction.

• *Property 4*: in this case data does not flow at all. Both properties can be updated locally, but never propagate or accepts any update from the outside.

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. Figure 4.5b shows the two possible cases in which permissions do not match: • the left component tries to publish the updated value of *Property 1*, but the message is discarded because the paired property inside component 2 did not subscribe to receive update notifications; • similarly in the second case the property on the right expects to be able to subscribe to property 2 inside the first component, but the property does not publish any update when the property is changed.

This approach requires both communication end points on different devices, potentially owned by different users, to agree that a property should be shared. This is important because components run on a Web browser and permissions can be unilaterally changed manually by malicious users.

These permissions make it possible to create common propagation patterns. In Figure 4.6 we show the application described earlier in Section 4.2 extended with the data flow direction annotations. Figure 4.6a shows the annotation at the component level. Similarly to the example without annotations (Figure 4.4a), all the components hold and synchronize the property values of the paired components even if they do not need to access them. With the annotations the flow of the data is limited and goes from the *presenter pad*, to the *presenter screen* and

Table 4.1. Storage mechanism chosen based on the sharing scope, component scope, device deployment and persistency of a liquid property.

Granularity		**Component level**		**Property level**	
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			
Persistency	Sharing Scope				

finally to the *viewer screen*. In case the students are able to hack into to the *screen view* property and change the value stored locally, the value is discarded and it is not synchronized inside the *presenter screen* component. In Figure 4.6b the same concept is applied to the properties instead of the components. The resulting data flow is identical to the flow described in the component level granularity.

4.4 Liquid Storage

In the previous section we discussed the data flow of a liquid application by abstracting where the data and state is physically stored by claiming that the data is stored locally either inside the components or by the properties. While it is true that the value of a liquid property is always **cached** inside the liquid component, at the same time it cannot be **permanently** stored inside it, because components can move and change location as the user moves from a device to another. Depending on the following four dimensions the developers can decide where to store the data of their liquid Web applications:

• **Granularity**: (see Section 4.2) the component scope defines whether the liquid data layer is synchronized between components or properties. The values of the granularity can either be: – Component level; – Property level;

• **Sharing Scope**: the sharing scope of components or of properties is an annotation that defines if the components or properties of a liquid Web application can be automatically paired by following a common sharing policy. The possible values of the sharing scope annotation are:

– **Global**: a global component or property is automatically shared with all in-

stances of every component or property with matching identifiers (e.g., they share the same constructor URI or name). Since those components and properties are automatically paired, it is not necessary to pair them explicitly when annotated. The state contained in a global component or property can be accessed from any connected device running the application. We encourage developers to minimize the usage of global components and properties as much as possible, in order to avoid that big portions of data are synchronized across multiple devices. In fact, this is exactly what we want to avoid with fine-grained component-based liquid applications for increasing privacy and for saving bandwidth and storage space;
– **Shared**: the components or properties with matching identifiers (e.g., they share the same constructor URI or name) are automatically paired in the liquid application;
– **Local**: the components or properties are never automatically synchronized with other components or properties. Nevertheless components and properties can still be manually synchronized with other components at runtime (e.g., through the LUE API primitives).

• **Device deployment**: the device deployment defines if components and properties can be paired across multiple devices. The possible values of the deployment are:
– one device: a component or a property can be paired with other components and properties which are deployed and instantiated on the same Web browser (e.g., same tab or another tab);
– many devices: a component or a property can be shared with any component or property, even if it is running on another Web browser (e.g., on the same or different device).

• **Persistency**: the persistence defines for how long a property should be stored. We distinguish the following values of persistency:
– **Persistent**: the state of a component or the value of a property is permanently stored even if all instances of the liquid components containing it are closed or all the devices connected to the application are shut down;
– **Session**: the state of the component or of the property is stored until the user closes every Web browser running instances of the annotated components or properties;
– **Volatile**: the value of the component or of the property is stored until at least one instance of a component or property holding the value exist.

The composition of these four dimensions makes it possible to decide where the state of a property should be stored within the liquid Web application. Depending on where the state is stored, the latency and bandwidth consumption of

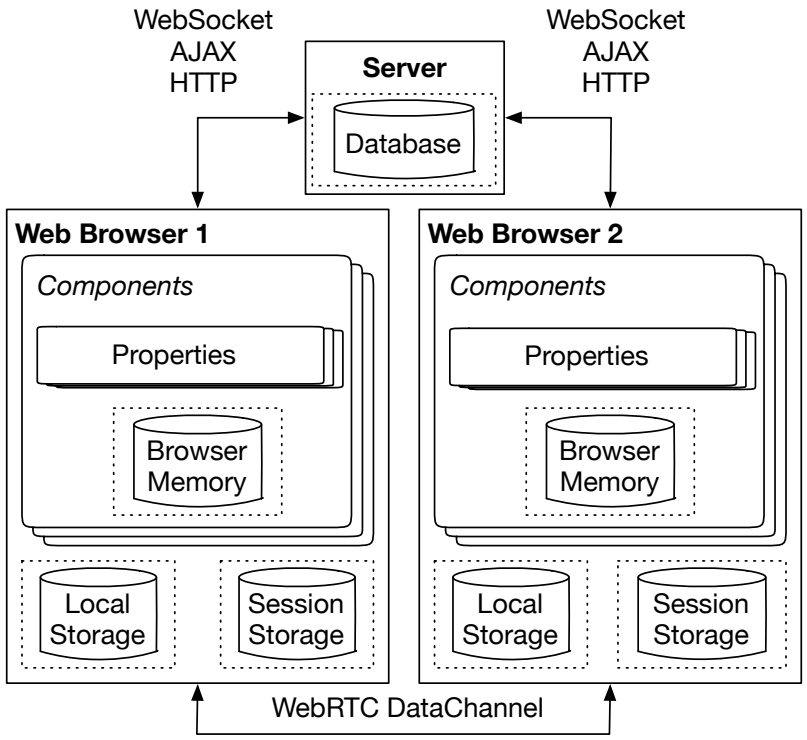


Figure 4.7. Level 4 liquid software architecture: storage deployment and flow of the data across two Web browsers and the server.

the synchronization can be lowered as much as possible.

We identify four possible deployments of the liquid state:

• **Browser Memory** (JS Heap): the Web browser memory is the best place to store volatile properties. Whenever the component is closed, the state of volatile components or properties is lost.

• **Local Storage** [Moz19d] allows to store and efficiently synchronize the state among all components and properties running on the same Web browser (e.g., same tab). If the HTML5 LocalStorage API is not available on the Web browser, the server-side solution is used instead.

• **Session Storage** [Moz19e]: unlike the local storage, the session storage discards the state when a *session* ends. If the HTML5 SessionStorage API is not available in the Web browser, the server-side solution is used instead.

• **Server-side Storage**: global components and properties potentially require all devices to access and synchronize their state. The perfect target for storing the global state is the signaling server which is directly connected to all devices running the application, and therefore to all instances of all liquid com-

ponents and properties. Moreover the server is the only component of the which survives when all clients disconnect, and thus can safely store the state of persistent global components and properties. The global state 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. When an application uses the signaling server for storing the global state lowers the maturity of the whole liquid application from level 5 to 4.

In Table 4.1 we summarize where the state of components and properties should be stored for each valid combination. Liquid components and properties are mapped to the corresponding storage mechanism based on the four dimensions discussed previously.

The liquid software should always try to minimize the usage of bandwidth when the data is synchronized and should try to lower the latency when migrating and loading the initial state of a component. In Figure 4.7 we show the possible architecture of a level 4 liquid application (with either component or property level granularity). The synchronization mechanism can be directly implemented in the liquid components where the value of a property is cached. Liquid components can directly exchange messages through WebRTC DataChannels in a multi-device P2P mesh if the data channels are available, or by using internal messages and events during single-device synchronization. The different data storage we described can used to make data persistent. Synchronization is carried out by the signaling server for global persistent properties, while shared properties are synchronized using the P2P mesh built with WebRTC among all devices.

Now that we presented the design of the liquid data layer, we can discuss the design of the liquid logic layer in Chapter 5. The design is built on top of the P2P communciation and synchronization and would not guarantee all liquid-related quality attrbitues if the the devices are not able to send messages between each other directly.

Chapter 5

Liquid Logic Layer and Liquid WebWorkers

In this chapter we focus on the logic layer as we discuss how to speedup the performance of liquid Web applications. Users interacting with multiple devices may trigger data synchronization activities that will ensure a consistent view over the state of a distributed liquid Web application. Having multiple, partially idle devices also opens up the opportunity to exploit their computational resources to speed up CPU-intensive tasks. In this chapter we focus on the business logic layer of the application and show how we can transparently offload the execution of CPU-intensive tasks among the active devices on which the application has been deployed. As opposed to vertical offloading which takes advantage of remote Cloud resources [111], here we introduce an horizontal offloading approach, where only local devices are involved. In our proposed solution the horizontal offloading is transparently handled by Liquid WebWorkers (LWWs), which take care of transparently migrating jobs across devices. In this chapter we present the design and API of LWWs.

Like standard HTML5 WebWorkers [Moz20d], also LWW are designed to perform background computations in a parallel thread of execution. Unlike standard HTML5 WebWorkers, the work can potentially be transparently offloaded across different devices. To do so, LWW use a simpler stateless programming model, which helps developers identify the boundaries of the task to be offloaded. LWWs receive discrete atomic jobs to be processed and produce the corresponding results all at once. The computational offloading is kept completely transparent from the developer, who can use specific task placement policies to prioritize the available devices according to different criteria.

5.1 APIs

LWW take care of executing tasks by invoking the corresponding HTML5 WebWorker [Moz20d]. LWWs are organized into a *pool*, whose goal is to manage their life-cycle, transparently choose on which machine the tasks should be executed, and reliably dispatch tasks towards the corresponding appropriate LWW, which can be located either locally or remotely.

The Liquid WebWorker Pool (LWWPool) and the LWW expose their own API that can be used by the developer for building multi-device liquid applications. Operations inside the LWWPool are executed asynchronously because they require to communicate with remote devices or exchange messages between the global JavaScript (JS) context and the worker. For this reason we decided to deal with asynchronous operations with Promises [Moz20c], which may invoke either a successful or a failing callback upon completion.

A rejected promise may return two types of error: either a *communication error* or an *execution error*. In the first case a failure happens during the offloading of a task from a device to another due to a problem in the sending process, either because there is no connection linking the two devices, because the remote machine is currently unavailable, or because a timeout triggered. The second error type is thrown whenever there is a problem with a LWW instance, either because the remote LWW is not yet instantiated or there was an internal error in the LWW execution.

5.1.1 Liquid WebWorker Pool (LWWPool) API

Table 5.1 lists all methods exposed by the LWWPool API. The LWWPool can be instantiated by passing the reference to a *sendMessage* function whose signature must accept two parameters: *deviceID* and *message*. This function will be called every time the LWWPool has to deliver a message to another device, it does not matter to the pool how the payload is delivered, but the pool expects that the function reliably delivers the whole *message* object to the device labeled as *deviceID*.

The LWWPool API exposes the following eight methods:

- **createWorker:** instantiates a new LWW and automatically binds it to the LWWPool. The pool may contain any number of workers, limited only by the Web browser WebWorker limit and the available resources. *WorkerNames* are unique (e.g., URI) if the pool is requested to create a worker with an already existing name, then it will fail and return a rejected Promise. The script can be either a *glsurl* pointing to a Web resource that can be fetched [Moz19g] with an HTTP

Table 5.1. Liquid WebWorker Pool API

Liquid WebWorker pool API	
Constructor LiquidWebWorkerPool(sendMessageFunction)	
sendMessageFunction signature sendMessage(deviceID, message)	
Method name and parameters	**Return value**
createWorker(workerName, scriptURI)	Promise(workerInstance)
getWorkerList()	Promise(workerNameList)
updatePairedDevice(deviceID, data)	Promise(deviceID)
removePairedDevice(deviceID)	Promise(deviceID)
callWorker(workerName, message)	Promise(response)
_callWorker(workerName, message)	Promise(response)
forwardMessage(message)	Promise()
terminateWorker(workerName)	Promise(workerName)

request, or it can be a String or a Blob [Moz19i] containing the actual script code. Both parameters are required.

• **getWorkerList:** this method returns a JS object containing all the references to the instantiated LWWs contained in the pool, indexed by the corresponding workerNames.

• **updatePairedDevice:** this method updates the information about the paired devices stored inside the pool. The *deviceID* is the same that will be passed in the *sendMessage* function whenever it will be called. The data is stored in an object that contains information about all devices. Depending on the policy rules employed, this object may contain different information (in Section 5.3.3 we discuss policies).

• **removePairedDevice:** this method removes a paired device from the stored list of paired devices. The LWWPool will take care of ensuring that any task currently offloaded on the removed device will eventually complete either with succeful or rejected promises. No new tasks will be assigned to the removed device.

• **callWorker:** the function is used to submit a task into the pool, which will be executed either locally or remotely. Once submitted, the pool decides where the task will be executed, then it creates the corresponding promises and calls the *sendMessage* function if the task is executed remotely, otherwise it will call the *_callWorker* function.

Table 5.2. Liquid WebWorker API

Liquid WebWorker API	
Constructor LiquidWebWorker(LWWpool, workerName, scriptURI)	
Method name and parameters	**Return value**
callWorker(message)	Promise(response)
_callWorker(message)	Promise(response)
terminate()	Promise(workerName)

• **_callWorker:** this method is used to submit a task into the pool for local execution. This method directly pushes the task message into the corresponding local LWW instance and waits for its asynchronous response by setting up a promise object.

• **forwardMessage:** whenever a device receives a message sent from another device after the *sendMessage* function is called, the LWWPool expects that the message is forwarded from the middle-ware to the remote pool by calling the *forwardMessage* function.

• **terminateWorker:** this method ends the life-cycle of a LWW instantiated inside the pool. If the *workerName* is invalid or undefined, it returns an error.

5.1.2 Liquid WebWorker (LWW) API

Table 5.2 lists all methods exposed by the LWW API. If an invalid or undefined LWWPool is passed as a parameter of the constructor, then the methods *callWorker* and *_callWorker* will behave equivalently and the LWW will never attempt to offload the execution to any remote device. Without being connected to a pool, the LWW cannot determine where the submitted tasks should be executed. The LWW does not store information about paired devices, nor it knows if it is paired to other LWWs as this information is managed by the associated LWWPool.

Developers can call methods on the LWW instances without the need to proxy their execution requests on the pool, since the LWW object itself exposes an API. The LWW exposes three methods:

• **callWorker:** this method submits a task into the LWW, if the worker is bound to a LWWPool then it will request the pool if the task should be executed remotely or not, otherwise it will automatically call the *_callWorker* method.

• **_callWorker:** this method bypasses the LWWPool policies and executes the

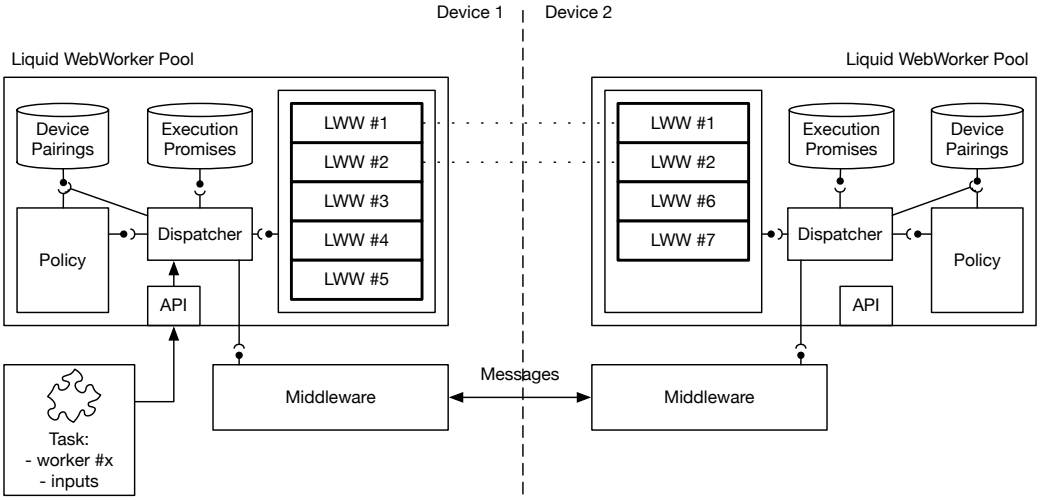


Figure 5.1. The LWWs architecture. Arrows show the flow of a task and the exchange of messages between clients. Dotted lines indicate paired relationships between LWW instances.

tasks directly on the issued worker locally.

• **terminate:** this function will terminate the WebWorker instantiated in the background, and safely delete all references pointing to the LWW instance. The termination is immediate and does not wait for the end of the task execution.

5.2 Design

Figure 5.1 shows the main components of the LWWPool running across two devices. Tasks can be submitted from either devices and the pool will decide whether they will be executed using workers of the local pool, or they will be offloaded to other devices. The middle-ware component in the diagram is outside the scope of the LWWPool and takes care to dispatch, receive and forward messages in behalf of the LWWPool to other devices. The middle-ware can be implemented with any technology or mechanism as discussed in Section 4.1.

In addition to the set of workers, the LWWPool stores references to the submitted and the currently executing tasks in the form of *pending promises*. It also maintains information about the *paired devices*:

• **pending promises**: for all submitted tasks, the pool creates a promise that waits for the worker to complete the computation and returns the results by passing the response and its associated unique identifier. The promise contains

the callback that must be fulfilled or rejected when the remote device or the local worker responds. The payload of the response contains the identifier of the corresponding promise, which can be easily retrieved from the corresponding dictionary stored inside the LWWPool.

• **paired devices**: the pool keeps track of all paired devices. This information contains the hardware specification of the devices, such as its type (e.g., desktop or phone) or any other information useful to the policy component for taking task offloading decisions (e.g., processor specifications, battery levels, OS versions).

The *dispatcher* component forwards tasks to the correct LWW and thus the correct device (e.g., local or remote). The decision on where the execution of the task will happen is controlled by the *policy* component, which uses data fed from the *device pairings* storage in order to take a decision. Whenever the dispatcher forwards a task, then it also saves the corresponding callback promise. When the task offloaded to a remote device, the dispatcher does not send the task directly to the remote endpoint, but it creates a new message and forwards it through the pre-configured middle-ware (see Section 5.1.1). Each message contains in its payload the corresponding *promise identifier*, the *inputs* of the task that need to be executed, and the *name* of the worker that must be invoked on the remote machine.

The dispatcher component can create new WebWorkers either by passing a URI pointing to a script stored in a central server, or by passing the content of the script as a String or as a Blob [Moz19i] Object that can be directly shared between devices without the need to fetch it from a Web server. If necessary, the dispatcher is able to instantiate the WebWorker script by converting the String to a Blob Object.

LWW are designed to be used for **stateless** computation; in fact, paired workers do not share or synchronize any data among each other. Likewise, every job is treated as an independent computation. Nevertheless it is possible to simulate stateful computations by submitting a task that would include as input the previous state of the worker, and then return the new state with the result so that it can be stored and passed along with the next task. This way, each task of the sequence can still be transparently sent to different devices.

The sequence diagram in Figure 5.2 illustrates the LWW call life-cycle and how the components inside the LWWPool communicate during local and remote execution. The assumption is that *device1* and *device2* have been paired and workers *w1* and *w2* have been created on both devices. In the example a task addressed to the worker named *compiler* is submitted by invoking the method *callWorker*. The pool will determine where the task will be executed by invoking

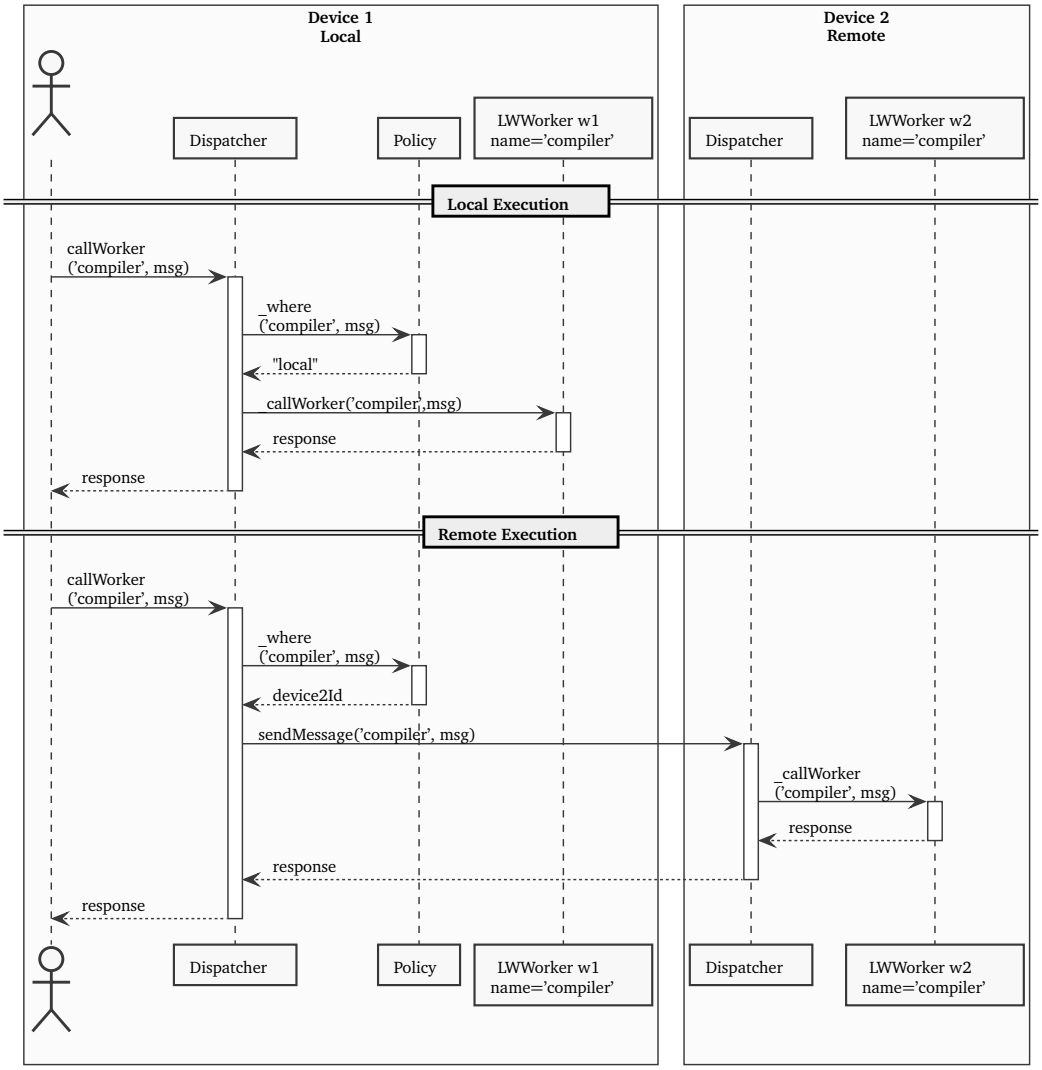


Figure 5.2. Local and remote execution sequence diagram.

the internal *_where* function of the the *policy* component. In the first case the *policy* component chooses to execute the task locally. This results in the local call to the corresponding LWW. The *response* is asynchronously computed within the worker and passed as a parameter in the fulfilled promise. Internally, workers use the standard HTML5 postMessage/onMessage API to exchange their input and output data with the LWWPool. This way, from the perspective of the caller, executing a task locally or remotely is indistinguishable.

In the next scenario the caller invokes the *callWorker* method a second time and eventually receives a response inside the fulfilled promise, however inside the pool the process sequence changes whenever the *policy* component chooses

to execute the task remotely. In this case the pool first sends a message to the remote device (by relaying the message through the middleware), the remote pool executes the task on a remote LWW and eventually it will send back a response. If no response is received within a given developer-configurable timeout, the LWWPool will attempt to find another device and resubmit the task. If eventually no more remote devices can be found, the task will be executed locally.

5.3 Features

The decision on where a task should be offloaded to, is taken based on different criteria and following constraints established by the liquid Web application developers, users or device owner preferences. To do so, in this section we outline a number of features that allow to enhance the flexibility and customizability of our LWW prototype.

5.3.1 Micro-Benchmark

In order to be able to implement valid policy rules inside the LWWPool, we need to predict what are the capabilities of each connected device. Running a macro-benchmark [92] on all the devices before they are allowed to join the liquid Web application would not rank the machines correctly. Macro-benchmarks test the performance of a whole system, however liquid applications are sand-boxed inside the Web Browser, which does not give full access to the device resources. For this reason we aim to assess only the resources accessible from the Web browser tab that is running the application. Moreover a macro-benchmark is an invasive process meant to be ran as stand-alone process in order to avoid interference from other non-idle processes. This would prevent users from interacting with their devices while the benchmark is running, which may take a long time to complete.

In our scenarios we need to be able to predict the capabilities of a device for as long as it connects to the liquid Web application. The amount of available resources provided by the device may dynamically change at runtime, because users can close or open new tabs while they are browsing the application. The benchmark should be repeated over time to accurately track the amount of available resources.

We decided to follow a micro-benchmarking approach [103], which is suited for mobile Web-enabled devices and allows us to test the performance of the ac-

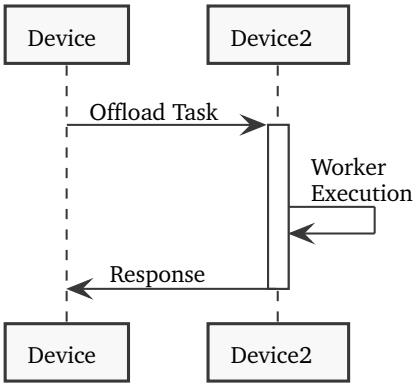


Figure 5.3. Task offloading without failures.

tive Web browser tab from within the browser itself by exploiting HTML5 standard APIs.

The LWW pool runs a micro-benchmark on startup after the pool is instantiated, then it keeps re-running the test at regular intervals. The interval time span is configurable by the developers of the liquid application. The benchmark runs in a dedicated background WebWorker and does not prevent the user from interacting with the application while it is executing. The benchmark environment is created with the library *Benchmark.js* [BD16] and the benchmark test-bed can be customized by the developer of the application.

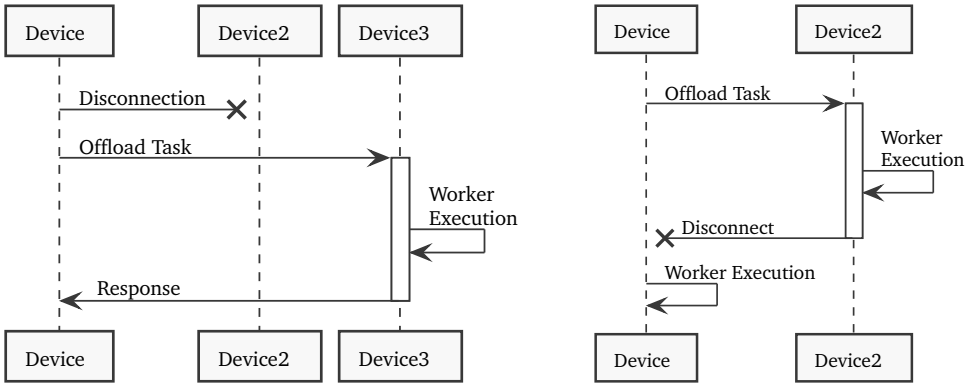
The result of the benchmark represents the average number of iteration per cycle it was able to perform during the execution. We use this number to rank our devices, making it possible to compare their performance. In Section 9.2 we evaluate the performance of the micro benchmark.

5.3.2 Failure Handling

During the task offloading process, failures may happen. The most common failure in distributed systems derives from disconnection of the peers [158], however failures may be generated also by faulty operations such as during task executions or faulty policy rules predictions [112].

In Figure 5.3 we show the expected sequence diagram of the offloading process for LWWs with synchronous data transfer: 1. *Device* offloads the *task* to *Device2*; 2. *Devices2* receives the *task*; 3. *Devices2* executes the *task*; 4. *Devices2* submits the response to *Device*; 5. *Device* receives the response and make use of it.

In this scenario we recognize that the offloading process may fail for three



(a) The device disconnects before the task is offloaded.

(b) The device disconnects after the task is offloaded, but before the response.

Figure 5.4. LWW failure scenarion: disconnection.

reasons:

- Failed connectivity - the communication between the two devices is interrupted during the offloading process;
- LWW failure - a run-time error during the execution of the offloaded task occurs;
- Timeout - the task does not complete within a given amount of time and the device does not send back a response.

In order to create a reliable system, in this section we propose a solution for all three scenarios.

A peer can disconnect anytime throughout the whole task offloading process, Figure 5.4 shows how the LWWPool recovers when a peer disconnects before the task has been completely sent to another device (see Figure 5.4a), and how it recovers when the task has already been offloaded before the peer disconnects (see Figure 5.4b). In the first case (Figure 5.4a) it does not matter if *Device2* disconnects before the starting device has chosen where to offload the task, after it choses the target or even during the task offloading process. Since the task execution has not started yet, the disconnected device will be excluded from the ranking of candidate devices and the second most-suitable device will replace it as the new offloading target. In the second case (Figure 5.4b) it does not matter if *Device2* disconnects before, during, or after the *Worker Execution* finished, in all three cases the starting device will immediately notice the disconnection and by default it will try to recover the execution by running the worker locally. The LWWPool can be configured to retry the execution on the next most-suitable device. If there are no other devices connected, the device will attempt

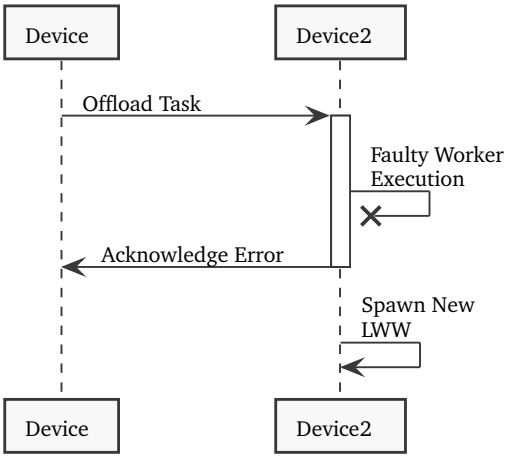
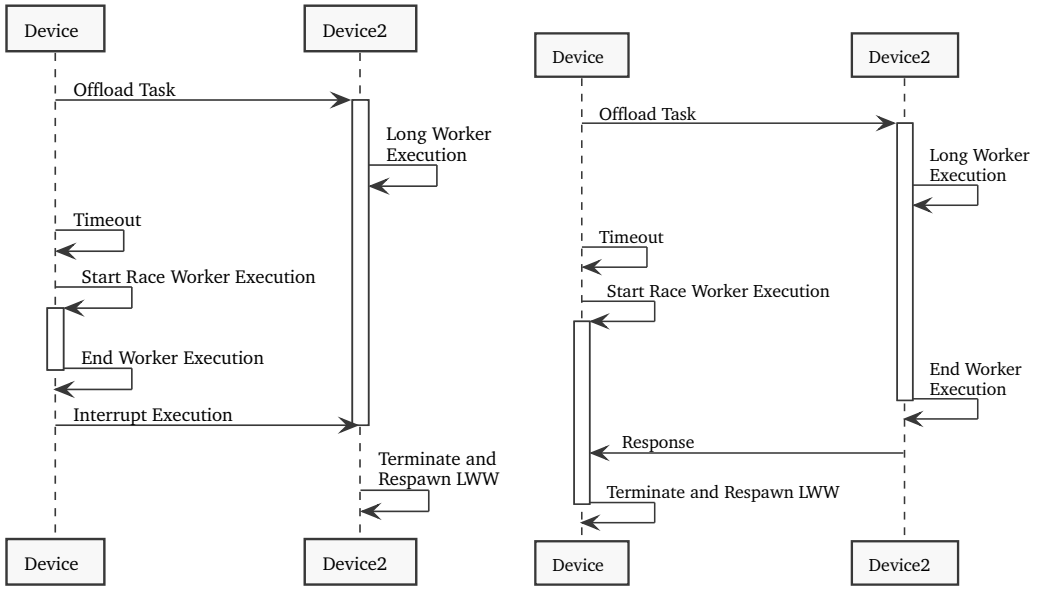


Figure 5.5. LWW failure scenario: run-time error during the offloaded task execution. The remote LWW is independently respawned but the local device should decide how to recover the task execution.

to run the execution locally.

Figure 5.5 shows how the LWWPool recovers when an error happens during the offloaded task execution, which may crash the LWW. The LWWPool is able to immidiately detect when a worker throws an execution error and reacts to it, and as a consequence it sends a message to the source device by acknowledging the failed task execution. The error response includes the reason of the crash and will not interrupt the execution of the main liquid application process on *Device2*. After the response has been sent, a new LWW will be spawned on *Device2*, which will return to be available to service other task offloading requests. By default the source device does not try to locally or remotely re-execute the failed task. The decision is left to the developer of the application that must define which recovery operation is executed by catching the error acknowledgement event from the LWWPool rejected promise.

In the last scenario no task execution response is sent by *Device2* back to the starting device within a given amount of time even if there are no problems with the connection (see Figure 5.6). This can happen because the LWWPool decided to offload the task to a slow device, or because *Device2* cannot complete the task execution before a timeout occurs. Whenever the timeout triggers, the starting device starts to execute the task locally. This creates a race between the local and remote task execution: if the local task ends before *Device2* has responded, then the starting device notifies *Device2* that it does not need its answer anymore, when *Device2* receives the message it will terminate and respawn the corresponding LWW. If *Device2* answers before the starting device finishes, then the opposite



(a) The starting device wins the race. (b) Device2 wins the race despite the timeout.

Figure 5.6. LWW failure scenario: timeout without disconnection and local task re-execution race

happens and the start device terminates and respawns the LWW.

Developers can set the default timeout as part of the LWW configuration and also associate a different timeout with each task. If the timeout is set to *zero*, then the LWWPool will always start a race between the local device and the remote peer. In this case the LWW will attempt to compute tasks with the highest speed among two devices, however it will also increase the energy consumption on both devices.

5.3.3 Task Offloading Policies

Policies are needed for making the LWWPool able of automatically decide where to execute tasks. This could be achieved by feeding the *policy* component with predefined rules selected by the developers of the liquid Web application, e.g., trade-off between energy consumption vs. performance.

Policy rules can impact in the overall execution time of an application and the developer needs to be able to enable or disable some rules depending on the context of the application they are building.

- **Battery status** [86] - in the Web browser it is possible to gain access to the

battery status of a device by using the HTML5 Battery Status API [Moz19j]. With the API it is possible to detect whether a device is currently charging or how much charge is left in its battery. The policy rule can exploit it to prioritize plugged-in devices over battery-supported devices. Tasks would be offloaded to devices with an higher charge level, which would decrease the energy consumption of devices with a low battery level.

• **Privacy or security constraints** - the users of a liquid application can interact with devices they do not directly own. Whenever the users interact with shared or public devices, they have to be aware that they are connected to other people's devices. In any situation where the users interact with any device they do not own, the developer should make sure that the users private data is not sent to a stranger device. The policy rule can decide to send data only to the devices they whitelisted, or to any device owned by a whitelisted user. Similarly, the users should be protected from receiving tasks from devices they do not trust (e.g., blacklisted devices).

• **Device types** - as a heuristic, when lacking additional information, the offloading decision can be based on expectations on the performance of a device by knowing its device type, such as *Desktop*, *Laptop*, *Tablet*, *Phone*. The policy rule would for example assume a desktop computer to be more powerful than a smartphone. The liquid Web application can infer the type of device from within a Web browser by checking the size of the screen or the user-agent of the device, which however may be changed by the users and would not give any direct evidence about the performance of a device [178]. Precise information about the underlying hardware is unfortunately hidden from within the Web browser. Thus, classifying device by their type only may result in incorrect offloading decisions and should be complemented with, e.g., a benchmark or some other statistics over some probed task execution times as described in Section 5.3.1.

• **Communication and computation time** - the policy component should consider the exchanged **data size**, the available **bandwidth** (both upload and download for asymmetric bandwidth) and the **latency** between the devices into the decision. This policy rule makes offloading decisions based on Equation 5.1, where the $Communication_{time}$ is defined in Equation 5.2.

$$Communication_{time} + Computation_{time}^{remote} \leq Computation_{time}^{local} \tag{5.1}$$

$$\begin{aligned} Communication_{time} = (Data_{size}^{out} / Bandwidth_{upload}) + \\ (Data_{size}^{in} / Bandwidth_{download}) + (2 \cdot Latency_{time}) \end{aligned} \tag{5.2}$$

While the $Data^{in}_{size}$ and the network parameters (*Bandwidth* and *Latency*) can be measured before taking the offloading decision[1], the $Data^{out}_{size}$ and the $Computation_{time}$ may only be estimated or learned based on the characteristics of the LWW script and the history of its past executions.

The features we described in this section allow to simplify some of the terms of the inequality. With the micro-benchmark results we can attempt to predict beforehand which devices have the lower $Computation^{remote}_{time}$ and $Computation^{local}_{time}$. If the remote time is lower, then we can analyze what is the communication cost of the offloading process.

Later in Section 7.4.2 we discuss our prototype of the LWW and introduce the asynchronous data synchronization built in the middle-ware. The asynchronous data transfer lowers the size of $Data^{in}_{size}$ and $Data^{out}_{size}$. In the best case scenario, where all the data is already stored on the remote device, $Data^{in}_{size}$ and $Data^{out}_{size}$ become negligible as only a reference pointing to the data can be sent.

5.4 Scenarios

LWWs can be used to improve the performance of liquid Web applications in simultaneous usage scenarios featuring the opportunity to offload local computations to remote devices owned by the same user or by multiple users (as long as the users trust one another and are willing to share their CPU/energy resources).

Within the simultaneous use case scenario we distinguish two usage categories:

- **Unrestricted** - in this category the environment is composed only by devices that volunteer to freely share computations with each other. All the devices agree that they trust all other devices and they can offload computations freely. The devices will also attempt to execute all offloaded tasks whenever they receive them and promise to return valid results.
- **Restricted** - in this category the connected devices only offer limited access and a lower degree of trust, in which they cannot always execute or exchange tasks between each other. Devices can be restricted from executing or offloading tasks for multiple reasons, e.g.:

[1]The $Latency$ can be measured while the devices exchange the connection handshake. The $Bandwidth_{upload}$ and $Bandwidth_{download}$ can be estimated only after some large messages are exchanged between the two peers as unfortunately the HTML5 Network Information API [Moz19k] is not advanced enough to return the exact values for the upload and download speed, but it can only describe the type of connection, e.g., *WiFi*, *cellular*.

– *privacy* - in order to guarantee data privacy in multi-users scenarios, the application may restrict to offload tasks to devices owned by strangers. In thi case, the offloading will take place only among devices of the same owner. When users bring only one single device to run the collaborative application, this device can be prevented to offload tasks to others devices, meaning it has to execute every task locally;
– *security* - arbitrary LWWs migration and execution can be used to push malware to the neighbour connected devices and they can be used to execute malicious tasks on other users' devices. LWW policies can be implemented in such a way they limit the offload execution of certain tasks (identified by their names) and thus prevent the execution of unknown tasks on unaware devices.
– *application dependencies* - restrictions can also be programmed to satisfy application specific requirements or dependencies, e.g., in an IoT scenario only some kind of devices can be entitled to receive offloaded tasks, because the tasks would need to access some specific sensor attached to the device.
More in general, we distinguish between **push** or **pull** restrictions. A specific device can be restricted from pushing tasks that need to be offloaded on other connected devices, or a specific device can be blocked from pulling tasks which have been offloaded from other devices. In the extreme case, it could be possible that a device can only offload tasks to other devices without ever accepting to run tasks offloaded from other devices (or vice-versa).

5.4.1 Single User Scenario - Editors (Image Processing)

LWWs can be used to speed up the process of applying computationally intensive image filters on the pictures displayed in a multi-device Web application (see Figure 5.7). In the example the liquid application is meant to operate on three different devices: • **a smartphone** - which is used to take pictures through the integrated camera sensor; • **a tablet** - which is used to browse the pictures and is used to select which filters should be applied to the images; • **a laptop or a computer** - which is used to display the pictures on a big screen.

The three devices run the same application simultaneously and they share the pictures between each other, e.g., whenever a picture is taken on the smartphone, it is transparently broadcasted to all other devices.

Without LWWs the tablet is in charge of computing and executing all possible edit operations selected by the user, while the smartphone and the laptop would be idle most of the time as they would only serve as input/output devices.

When LWWs are activated, the devices are able to offload computations among one another. In this case the tablet does not have to be burdened with all

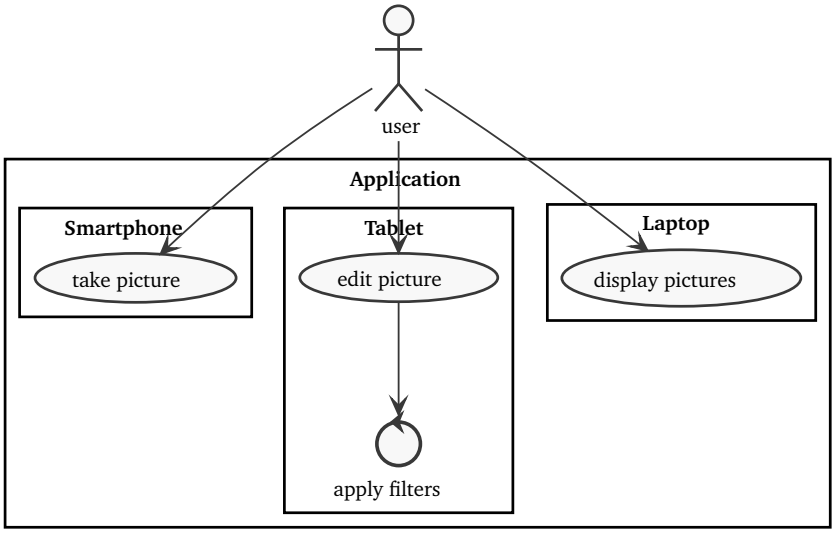


Figure 5.7. Image processing scenario with LWWs.

the image processing tasks, but also the smartphone and the laptop can participate with the goal of improving the overall response time of the image filtering feature of the application. In this particular scenario all the devices are owned by the same user and they are *unrestricted*, that means that any device can freely accept all incoming offloading requests and it can forward them to any other device. Offloading the filters to the laptop can save the battery energy of the tablet.

5.4.2 Single User Scenario - Public Displays

In Figure 5.8 we show a scenario where a single user runs an application on multiple devices, however not all of the devices are owned by the user. In this case the user owns the smartphone, while the public display is owned by the city, which deployed it outside of a train station. The display can be used by anyone by scanning a QR-code printed on the frame of the screen so that the encoded URL is opened on their device mobile Web browser. Once the phone is connected to the screen, the user can search on the displayed map for any interesting place in the city, or even compute the shortest path to a given location. While searching for a building is not a complex operation, computing the shortest path may take some time and, since the display should always be responsive to the user interaction, the computation for the shortest path is offloaded to the smartphone. In this use case scenario there is a clear trade-off between the execution time of the algorithm and the bandwidth required to send the map to the smartphones. In

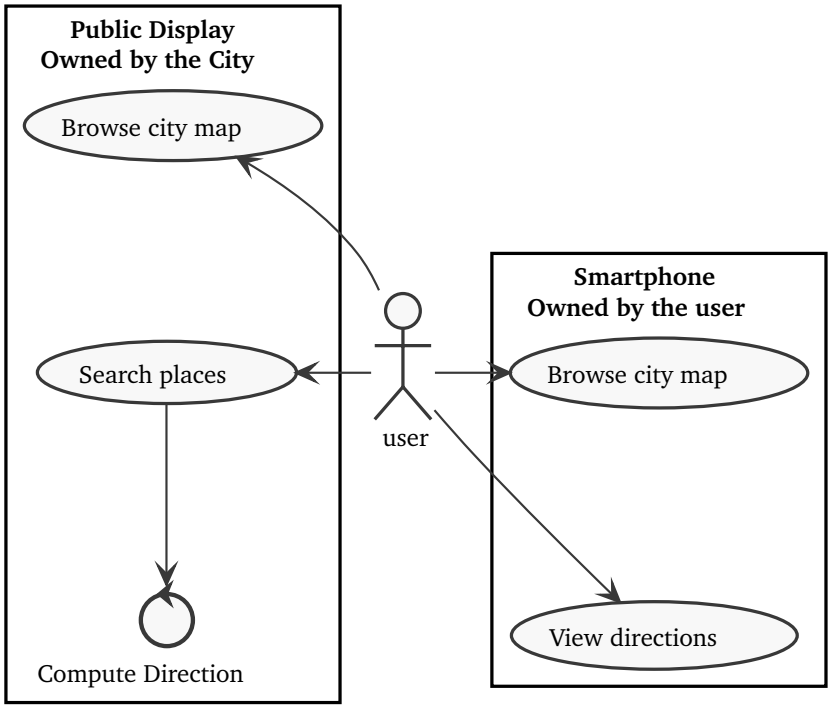


Figure 5.8. Public display scenario with LWWs.

the case that the display owners consider the execution time on the display more costly than the bandwidth usage, then they will prefer to offload the execution on the smartphones, even when the phone CPU is weaker than the one on the smart display. They are likely to choose the asynchronous data synchronization (discussed later in Section 7.4.2), as it will cache the map on the smartphones and it will make it available on the devices for multiple consecutive computations of the shortest route.

While users wait for the task completion, they can browse for other locations or even queue up new computations on their smartphone. Once the requested shortest path tasks are computed, the solutions are stored directly on the phone and the user may display them at any time, even if the smartphone is not connected to the public display anymore.

5.4.3 Multiple Users Scenario - Education/Teaching Programming

In this multi-user scenario we have two user categories: the students and the professors. The professors run the application on their own computers, while the students can access it with their laptops, or even with their tablets or smart-

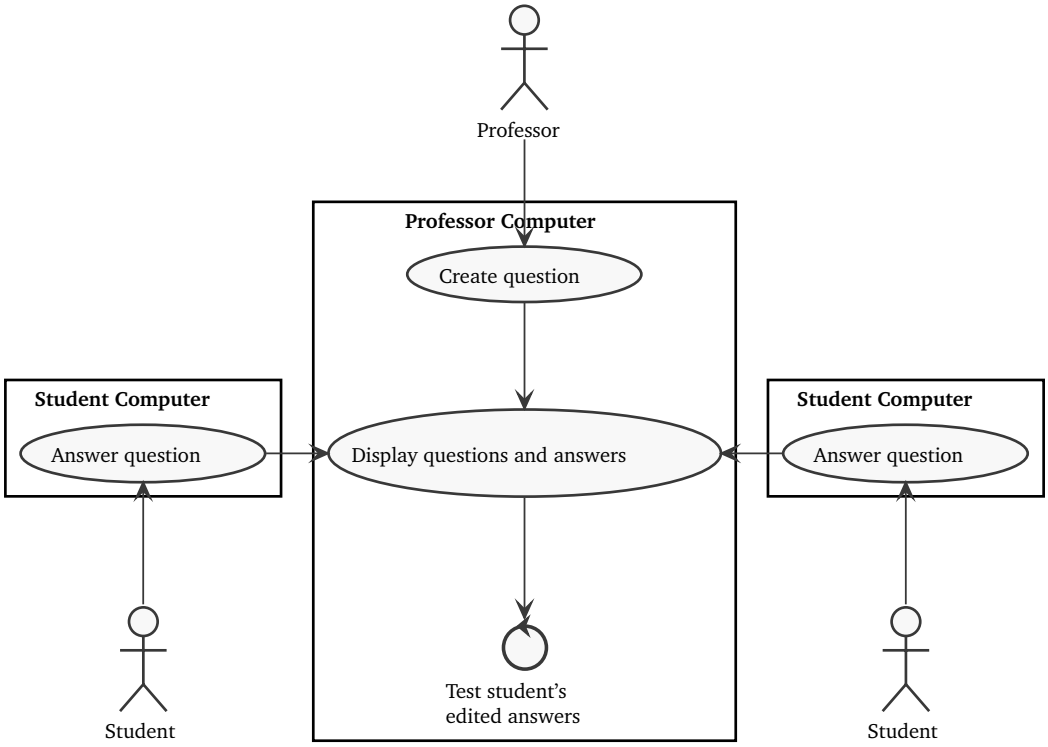


Figure 5.9. Education scenario with LWWs.

phones [175] (see Figure 5.9). The professors can create new questions at any time, e.g., "*transform the for-loop code into a while-loop code*" or similar programming-related questions. The students can see the questions and they can answer by sending a piece of code to the professor. The professors can then choose to display any received answer, they can edit the answers if they spot some errors and then they can display the result of the code execution returned by re-running the code. In order to display the result, the professors need to execute the code, which may lead to three main problems:

- the execution never finishes or it takes too long to finish;
- the code is malicious and tries to block the professor's computer;
- the code is malicious and attempts to corrupt the data contained in the professor's browser storage, e.g., it tries to display on the screen some private data, or it tries to communicate with external websites;

Without LWWs the code submitted by students must be executed on the professor device, with all the risks of executing buggy or malicious code that can stop the application or disrupt the lesson taught by the professor. In this scenario, LWWs are useful to offload the computation on the computer that originally sub-

mitted the answer. In this case the professor computer usage is restricted, because it does not accept any incoming execution request, but it always offloads them to the students' computers.

In the next chapter (Chapter 6) we finish the presentation of the design of liquid software by discussing how to make the view layer of an application liquid.

Chapter 6

Liquid View Layer and Liquid Media Queries

In this chapter we focus on the view layer as we discuss in detail *liquid media queries*, an extension to standard CSS3 media queries [56] that allows the developers to create their own CSS style sheets that get be activated when their Web applications are deployed across multiple devices. While as part of the LUE, end users can control which UI components are deployed on each device (e.g., by swiping or drag and dropping), developers can use liquid media queries to declaratively describe how their applications can automatically react to changes in the execution environment.

The developers of liquid applications should be able to offer to the users an automatic rule-based deployment mechanism for populating all of the users' devices with all the pieces of the application they are running, because a misuse of the manual LUE primitives may lead to non-intuitive deployments which contradict with the developer expectations and intent. For example, in the case of a picture sharing application, it should be possible to constrain the component in charge of taking and selecting pictures to smartphones, while the picture viewer component should be deployed on a device with a larger display. This way, users can select which picture to display from their personal smartphone photo library and take advantage of a public device to have a shared slideshow.

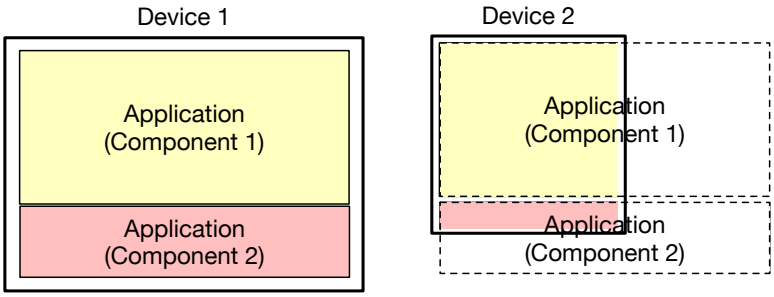
Choosing the appearance of a Web application and deciding how it should dynamically adapt to the devices it is deployed on, is a mandatory task during the design of a Web2.0 application [144]. *Responsive design* is the commonly followed best practice used to create UIs able to adapt to the devices' specifications [125]. Responsive design requires developers to decide how the UI is presented to the user and how it changes when deployed on different devices

with distinct input/output capabilities. The challenge of responsive Web design is to be able to adapt any Web application to any kind of Web-enabled device, ranging from small and weak smart objects, to the largest and more powerful computers connected to big screens [99]. While in the past designing responsive Web applications was difficult, nowadays we can easily design responsive Web applications with CSS3 and HTML5, which are the current standard used for creating responsive Web UIs. Nevertheless, following the birth and evolution of the IoT during the last decade [6], developers face new challenges that responsive Web design cannot solve on its own. Responsive UIs are meant to adapt to a single device at the time, however, as the number of devices owned by a user increases [Glo17], developers need to develop Web applications which can adapt their UI taking into account the whole set of multiple, heterogeneous connected devices (see Figure 6.1).

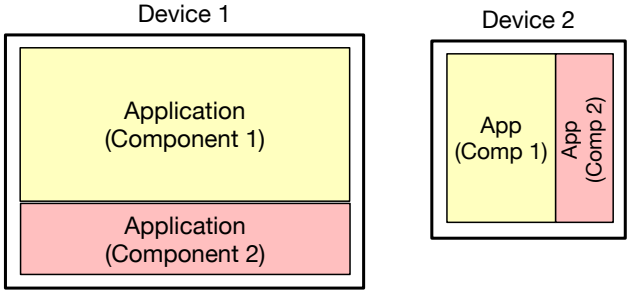
In particular, the goal is to allow developers to create their own complementary view adaptations, in which the users can take advantage of all their simultaneously connected devices. A complex UI can be scattered and presented on multiple devices, in such a way that its users can have immediate access to more information in comparison to single-device usage scenarios [2]. In fact, with the design of a complementary view, we have the opportunity to exploit companion devices and use them to extend the screen size to display parts of the UI of an application which would not normally fit the visible area of a single screen.

The multi-device adaptation needed for creating complementary views can be decomposed in three essential sub-tasks:

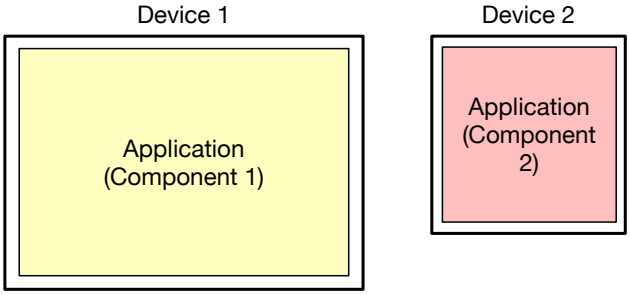
• **Adapt styles in a single device deployment**: whenever the whole application or a single component is deployed on a device, its UI needs to adapt. The appearance of the deployed software changes because of the device hardware specifications (e.g., the screen size), or because the user can interact with the application using different kind of interactions more suitable to the device hardware (e.g., *swiping* on a smartphone). Consequently some functionalities can be enabled or disabled depending on the device capabilities (e.g., *geolocation* on location-aware devices). Considering nowadays Web applications, the single device adaptation is already possible with the help of standard HTML5 and CSS3 [56].
CSS3 media types and features can be used to adapt the UI of an application to multiple devices by associating a CSS style sheet with some expected device characteristics. Standard media features consider qualities of the Web browser and its environment (e.g., the screen size and resolution, the output media, and the device orientation). If the media query matches what the device supports, then the corresponding style sheet is activated.



(a) Static View / No Adaptation: the Web application is meant to fit on a single device (Device 1) and when it is deployed on another one, it does not adapt. For example, if the screen is smaller (Device 2) than the one it was originally designed for, the user must scroll to see the entire application.



(b) Responsive View Adaptation: the Web application can adapt to different device capabilities.



(c) Complementary View Adaptation: multiple devices can be used concurrently to run the Web application which scatters and adapts the UI across multiple devices. In a complementary view, each device displays different components of the same application.

Figure 6.1. View adaptation options: Figure 6.1a - no adaptation with static definition of the appearance of the Web page; Figure 6.1b - responsive view adaptation; Figure 6.1c - complementary view adaptation

Standard CSS3 media queries are at the foundation of responsive UIs that adapt to a single device at the time.

• **Adapt styles in a multi-device deployment**: the latest CSS3 standard lacks sufficient expressiveness to describe the UI adaptation in a multi-device environment. CSS does not yet define media types and features that can be used to describe a multi-device deployment, therefore we cannot use it to describe multi-device views that can change styles whenever the user is using multiple devices simultaneously (e.g., react when the user connects a new device, or disconnects a previously connected device). Nevertheless, CSS3 is a well-rooted tool in the Web and we believe that its expressiveness for single device adaptations is very powerful, therefore we decided to extend it. As we are going to show in the next section, the concepts of *style sheets*, *media features* and *media types* can be used also for adapting the UI of an application to multi-device deployments and in Section 6.1 we define our own liquid media features and types. With the definition of new CSS3 media types and features we can dynamically change the styles of an application at runtime and react in real time to any change of the set of connected devices.

• **Automatically migrate components of an application between devices**: since our goal is to build fine-grained complementary view adaptations, we must be able to deploy and migrate pieces of an application among the set of devices. To do so, we need to define policies that can check the current deployment and decide whether the components need to be migrated every time the set of connected devices changes. The migration and deployment is not designed to be part of the liquid media specification we propose and in this dissertation our prototype (see Chapter 7) provides the actual migration mechanisms and LUE primitives. The prototype uses the liquid media queries to infer the multi-device deployment (in Section 6.2 we described how the prototype can infer it).

6.1 Automatic Component Style Adaptation

In order to implement the multi-device adaptation, first the application must be aware of when it is deployed on multiple devices. Additionally, it should react when the deployment configuration changes. Since standard CSS3 media queries do not define media types and features that can be used to define multi-device deployments, in this Section we introduce and describe new media types and features suitable for liquid Web applications (see Table 6.1). They can be used by the developers to define cross-device UI adaptations by declaratively constraining on which devices the components should be deployed on, and by controlling

Table 6.1. Proposed media types and features for liquid media queries.

Name	Description
Features	
`liquid`	Shortcut for `min-liquid-devices: 2`.
`liquid-devices`	The number of connected devices.
`liquid-users`	The number of connected users.
`liquid-device-ownership`	Whether the device is `private`, `shared` or `public`.
`liquid-device-role`	The application-specific role of a device.
`priority`	Redistribution feature (see Section 6.2.1)
`clone`	Cloning feature (see Section 6.2.2).
Types	
`liquid-device-type`	The type of device(s) running the application.

which style sheets should be enabled depending on individual properties of the set of devices connected to the application.

We define the following liquid media features and types (see Table 6.1):

liquid and liquid-devices - in *parallel screening* scenarios liquid applications are deployed on multiple devices in parallel. Detecting whether the liquid application is currently running on multiple devices is therefore required for the adaptation. The *liquid* feature refers to any deployment with at least two connected devices, while the *liquid-devices* feature makes it possible to create different views for specific numbers of connected devices. Similarly to CSS3 media queries, it is also possible to define the minimum and maximum values for the *liquid-devices* feature by setting the values for *min-liquid-devices* and *max-liquid-devices* (e.g., `min-liquid-devices:3` can be used to dynamically change the view of the liquid application when there are at least three connected devices).

liquid-users - in *multi-user parallel scenarios* the liquid application is deployed across multiple devices and multiple users can interact with it at the same time. The *liquid-users* media feature allows to adapt a UI depending on the number of users connected to the application. The features *min-liquid-users* and *max-liquid-users* can also be used for creating styles for single user applications (e.g., `max-liquid-users: 1`) and for multi-user applications (e.g., `min-liquid-users: 2`).

liquid-device-ownership - the types of access granted to the devices can be either *private*, *shared*, or *public*. A *private* device is owned and used exclusively by one single user. *Shared* devices are owned by one user, but they can be used

by another. *Public* devices (e.g., public displays [135]) can be used by both registered and authenticated users or by anonymous guests.

liquid-device-role - the *device role* is used to classify devices according to application domain-specific features. Developers can declare which roles they expect the devices used to deploy their application should play (e.g., *controller*, *console*, or *multimedia display*). Users can assign one of the predefined *roles* to their actual devices. To do so, the device-role property can be used to assign styles to be activated on devices with the assigned role. When the developers decide to use the *liquid-device-role* feature, the connected devices must be configured at runtime and a role must be assigned to them. The role metadata associated with the device can change at any time.

priority and **clone** - these two features are used by the liquid Web application to infer the redistribution deployment of the components. The description and possible values of these two features are discussed in Section 6.2.

liquid-device-type - the latest standard CSS3 media types only distinguish between *screen*, *print*, and *speech* devices. Depending on the context of the application, it can be useful to have a more fine-grained distinction of the kind of *screen* devices connected, so that they can be assigned to perform certain kind of tasks (e.g., desktop computers are used more for working in an office) [102], while other devices are more convenient in certain social situations (e.g., smartphones as opposed to laptops are more convenient during meals) [97]. In the current implementation of the prototype described in Chapter 7 *liquid-device-type* can be set to *Desktop*, *Laptop*, *Tablet*, *Phone*.

Listing 6.1 shows an example of the definition of a Web component which defines multiple liquid media queries. The component named `component-example` contains a *style* tag with two CSS3 media queries. These queries both use the `liquid-device-type` and `min-liquid-devices` features we have previously defined:

• the first liquid media query *liquid-devices:2* activates the style it encapsulates only when the set of connected devices consist of exactly two devices. Whenever there are two connected devices, the background color of the component is changed to red (see Figure 6.2);

• the second liquid media query *min-liquid-devices:3* reacts whenever there are three or more devices connected. As soon as a third device connects, the background color of the component is changed to blue, if more devices connect the color does not further change and remains blue (see Figure 6.2).

Listing 6.1. Component defining a style containing two liquid media queries.

```
<component-example>
 <style>
  @media (liquid-devices: 2) {
   :root {
    background-color: red;
   }
  }
  @media (min-liquid-devices: 3) {
   :root {
    background-color: blue;
   }
  }
 </style>
 <template> <!-- Component HTML --> </template>
 <script> /* Component logic */ </script>
</component-example>
```

6.2 Component Deployment Redistribution

In the previous section we introduced the media features and types that are needed to describe the multi-device environment, in this section we describe the policies that can be used to control the deployment of components among the set of connected devices. It is important to note that the deployment is not static, since the environment can change at runtime (e.g., devices can connect and disconnect while the liquid Web application is running). Whenever there is a change in the set of connected devices, a new deployment configuration is computed and the components are migrated across the devices accordingly.

Different policies can be used to decide where components will be migrated and the decision on how the components are redistributed across the devices is left to the developers of the Web application. The developer can choose the policy of the redistribution considering the following two different assumptions:

- **Redistribution only** - the application does not create new instances of a component during the redistribution, meaning that the number of instantiated components of the UI of the application remains constant;
- **Redistribution and cloning** - the redistribution of the UI allows to spawn new clones of existing components. In this case a given component can have additional instances spawned on suitable devices. When such devices disconnect, the cloned components are not migrated to other still connected devices, unless

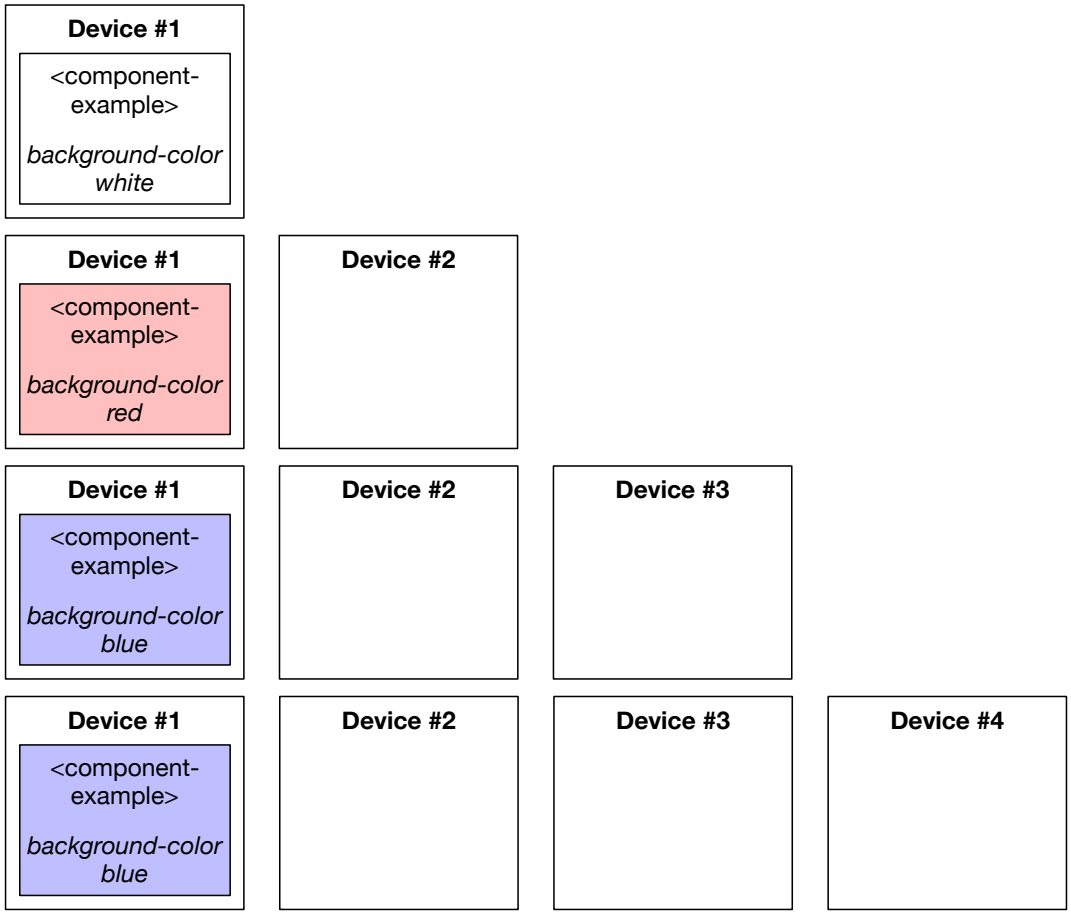


Figure 6.2. Style adaptation of the component described in Listing 6.1 when up to four devices are connected to the application.

it was the last instance in the set of devices.

The choice between the two assumptions is application specific and depends on how the developers expect the application to be used. The *redistribution only* assumption is more suitable for single-user applications running on a limited selection of devices, while *cloning* can be used for multi-user applications in which a certain component should be displayed on multiple screens, e.g., on each device of a specific type or on some device of each distinct user.

6.2.1 Redistribution step

How can developers control the target devices on which components should be deployed on? Developers can use the liquid media features and types we de-

scribed earlier. Whenever the developers define a liquid media query inside a component, the liquid application can assume that the developers are hinting that the component should be deployed on a device with matching features and types, if it is already available in the set of connected devices, or migrated on it if it becomes available while the application is running.

The decision on how to redistribute the components can be based on the following policies:

• **Exact match**: this policy decides to move a component to any device that matches all the constraints defined by all the liquid media queries defined in the component. If there is no such device, then the migration does not occur.

• **Maximize device-component constraint affinity**: each component can define multiple liquid media queries and it is possible that the developer chooses to create alternative media queries that cannot be accepted at the same time (e.g., in the example presented in Listing 6.1, it is impossible that both liquid media queries can be matched, because they hold exclusive values). The reason for the developers to design such liquid media queries is for adapting the same component to alternative deployment configurations. When the component media queries target different devices, it would be possible to migrate the component to any of those devices if they are available. If more than one such device is available, this policy migrates the component to the device that matches the most liquid media queries defined in the component, instead of migrating it only when all queries are exactly matched.

• **Priority-based**: the priority-based policy can control which device becomes the target of a migration when multiple devices match the same liquid media query. With this approach the developers associate a *priority score* to their liquid media queries. This policy moves the components to the device that accepts at least one liquid media query defined in the component, and in case there are more devices that accept the same query, then the component is moved to the one that has the highest priority. Listing 6.2 shows how the developers can define the priority of a liquid media query by assigning a value to the `priority` feature. In the example, the first liquid media query defines a style that should be activated with higher priority in respect to the second liquid media query. Figure 6.3 shows how the component defined in Listing 6.2 is moved to different devices when new targets become available. The component is initially deployed on a laptop, if a new phone or tablet connects, since the component defines at least one matching liquid media query, it migrates to either one of them and changes the background color accordingly. When both phone and tablet connect, then the component is moved to the phone and the background color is set to blue, because the priority defined in the liquid media query that matches the phone is

Listing 6.2. Liquid media queries including the `priority` feature.

```
@media (liquid-device-type: phone) and
       (priority:2) {
  :root {
    background-color: red;
  }
}
@media (liquid-device-type: tablet) and
       (priority:1) {
  :root {
    background-color: blue;
  }
}
```

higher than the one of the tablet.

• **Minimum number of components per device**: all the policies explained before do not always take full advantage of all connected devices, because multiple components can be migrated to a single device matching multiple liquid media queries, instead of scattering them among all available devices. This policy is primarily meant to work in conjunction with the previous policies as it always tries to instantiate at least one component per device, if there are enough components to be scattered in the set of connected devices.

• **Minimize migration cost**: if the set of devices changes often, it is possible that the redistribution moves many components around in a short amount of time. The result is that components may flicker between devices and thus hinder the usability of the liquid application. This policy minimizes the number of migrations when there is a change in the set of connected devices by ensuring the stability of the configuration (e.g., components are only migrated if the device on which they are running on is disconnected or if more suitable devices are connected, but are not shuffled between existing devices). Developers can also configure the policy to specify an upper limit to the number of migrations that can be performed during each adaptation.

The redistribution step deals with three possible outcomes and some of the policies we presented are more suitable than others depending on the scenario:

• $\#components < \#devices$: when there are more devices than components, some of the devices will not be selected as targets of the migration. In this scenario the *exact match* policy is useful to select the best device to deploy the components given a huge selection of different devices. Together with the

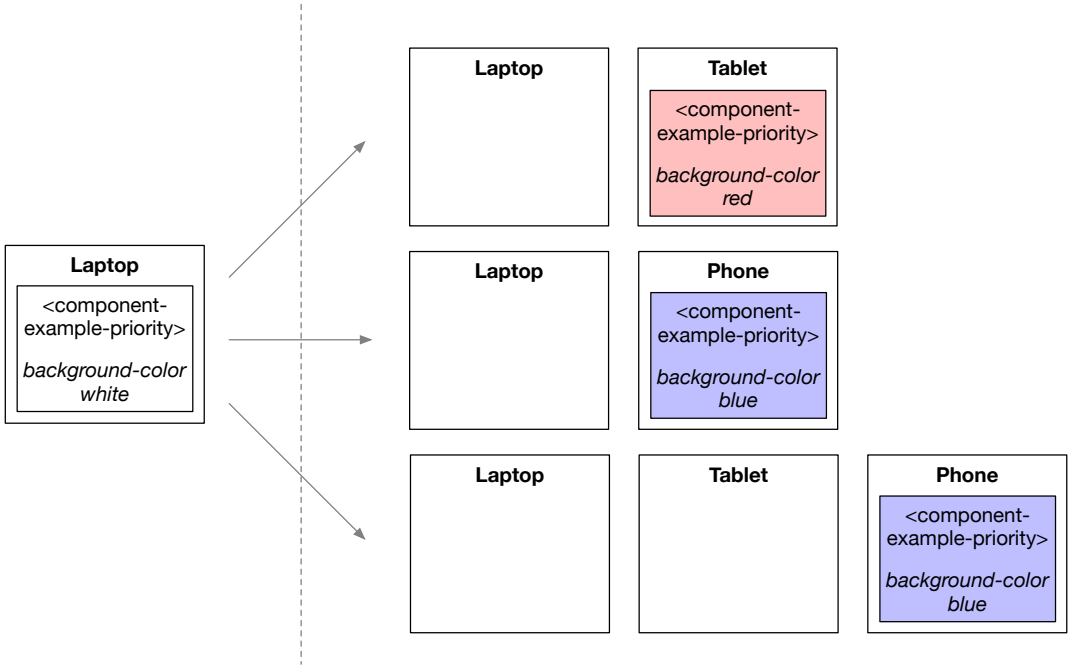


Figure 6.3. Redistribution of the component described in Listing 6.2 when it is initially deployed on a laptop and then new devices connect.

minimum number of components per device policy, the redistribution can target the best subset of devices desired by the developers.

- $\#components > \#devices$: in this scenario the component instances outnumber the devices, therefore multiple components are co-located on the same device. The *maximize device-component affinity* and the *priority-based* policies can be used to select the best configuration of devices for running the application when a small selection of devices is available. Again the *minimum number of components per device* policy can be used to avoid that the application is deployed on a single device when there are no matching devices available.

- $\#components == \#devices$: in this scenario the *minimum number of components per device* policy will instantiate a component on each device, taking full advantage of the set of connected devices. Given the small selection of devices, the *priority-based* policy is well suited for this scenario, since the components with the highest priority value are selected to be moved to the best matching devices first. In this specific scenario, when the developers choose to use the *minimum number of components per device* policy and the set of devices changes, it is possible that the redistribution completely changes the deployment of the application, which is not good in terms of user usability. In this case the *minimize*

Listing 6.3. Liquid media query including the `clone` feature

```
@media (liquid-device-type: phone) and
       (clone:*-phone) {
  :root {
    background-color: red;
  }
}
```

migration cost policy can be used.

6.2.2 Cloning step

The cloning step is independent from the redistribution process and it happens after the redistribution ends.

When multiple instances of the same component need to be deployed on multiple devices, developers must define an additional feature labeled `clone` within the liquid media queries. The clone feature enables multiple instances of the source component to be cloned across multiple devices instead of just migrating it on one of them.

In Listing 6.3 we show a liquid media query that defines the `clone` feature. In this particular case the component will be instantiated on all connected phone devices. The `clone` feature accepts values in the form of $N - feature$, where N is a positive non-zero integer or the symbol $*$, and $feature \in \{user,\ device, phone, tablet, desktop, laptop, shared, public, private, role = X\}$. The value N specifies the maximum number of instances of the source component which should be cloned across the set of available devices which match the liquid media query constraints in relation to the chosen $feature$. Their combination allows to write cloning rules such as:

- **1-user**: the component is cloned at most once per user, picking any of their available devices;
- **1-device**: the component is cloned at most once per device type;
- **2-tablet**: up to two component instances are cloned among all available tablets;
- ***-public**: the component is cloned once on each available *public* device.
- ***-role=dashboard**: the component is cloned once on each devices playing the dashboard role;

The `clone` feature works in conjunction with the other features and types of the liquid media queries, therefore a device matches the cloning feature only if

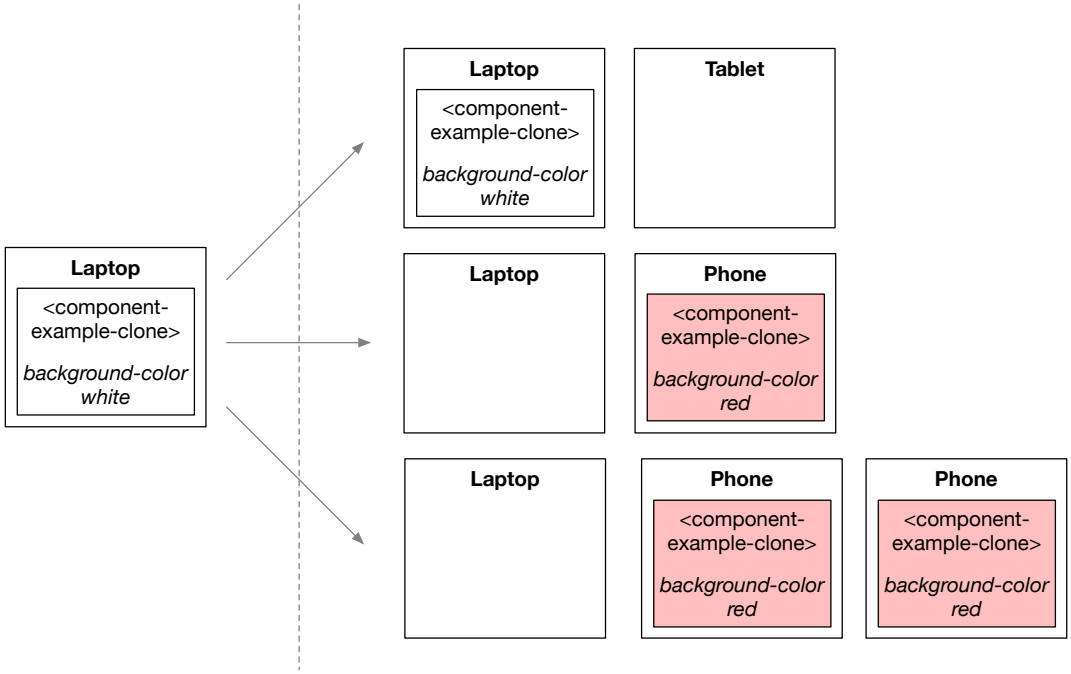


Figure 6.4. Redistribution and cloning of the component described in Listing 6.3 when it is initially deployed on a laptop and then new devices connect.

it also matches the whole liquid media query.

In Figure 6.4 we show how the redistribution and cloning of the component described in Listing 6.3 happens. The component is initially deployed on a laptop and is not migrated, nor cloned when the tablet connects, because the liquid media query does not match with the tablet. When a phone connects, then the component is migrated and if additional phones connect, then the component is also cloned on those devices.

6.3 Liquid UI Redistribution and Cloning Algorithms

The UI adaptation algorithm operates on three distinct phases: *constraint-checking and priority computation*, *redistribution and cloning*, and *local component adaptation*. The algorithm first decides which devices are suitable for displaying a component encapsulating the liquid media queries, then it migrates and clones the component on the highest priority device and activates the corresponding style sheet as soon as the component is instantiated on the target device.

6.3.1 Phase 1: Constraint-Checking and Priority Computation

The constraint-checking phase decides if there is a suitable device in the pool of connected devices that satisfies the liquid media query expressions encapsulated inside the components.

Algorithm 6.3.1 computes the matrix of valid target devices in which at least one liquid media expression is accepted. The matrix has size $\#components \times \#devices$. Each element represents with a positive integer the highest *priority* value of all the accepted liquid media queries encapsulated in the component, or *zero* if there are no accepted queries.

The matrix shown in Equation 6.1 is the *priorityMatrix* produced by Algorithm 6.3.1 during the example scenario we present later in Figure 9.11, when both $UserA$ and $UserB$ are connected. There are four instantiated components and seven devices connected to the application. c_{video}'s liquid media queries (see Section 9.3) are accepted by device d_{laptop}, d_{tv}. At least one query of priority *2* was accepted by device d_{laptop} and at least one query of priority *4* was accepted by devices d_{tv}. d_{phone1} accepts at least one query encapsulated in components $c_{videoController}$, $c_{suggestedVideo}$, the first one with priority *2* and the latter with priority *1*.

$$\begin{array}{c} \\ c_{video} \\ c_{videoController} \\ c_{suggestedVideo} \\ c_{comments} \end{array} \begin{array}{c} \begin{array}{ccccccc} d_{phone1} & d_{phone2} & d_{phone3} & d_{tablet} & d_{laptop1} & d_{laptop2} & d_{tv} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 2 & 2 & 4 \\ 2 & 2 & 2 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array} \tag{6.1}$$

$$\begin{array}{c} \\ c_{video} \\ c_{videoController} \\ c_{suggestedVideo} \\ c_{comments} \end{array} \begin{array}{c} \begin{array}{ccccccc} d_{phone1} & d_{phone2} & d_{phone3} & d_{tablet} & d_{laptop1} & d_{laptop2} & d_{tv} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 2 & 2 & 4 \\ 2 & 2 & 2 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \end{array} \tag{6.2}$$

Algorithm 6.3.1 also computes the *cloneMatrix* shown in Equation 6.2, which has a similar structure to the *priorityMatrix*, but stores only the information about the components that define at least one clone rule in the attributes of the *liquid-style* elements they encapsulate.

Algorithm 6.3.1 runs whenever one of the following *events* occurs:

- **A component is created or deleted from a device** - creating or deleting a

Data: Input: $priorityMatrix$, $cloneMatrix$
Data: Shared global state:
$components, devices, users, deviceConfigurations$
Data: Event
if *Event == component c created* **then**
  Add a new row in the *priorityMatrix*;
  forall $d \in devices$ **do**
    forall *liquid-style in the created component* **do**
      Check if the *device* accepts the liquid-style and save the highest priority in $priorityMatrix[c][d]$ and in $cloneMatrix[c][d]$;
    end
  end
else if *Event == component deleted* **then**
  Remove the corresponding *component* row from the *priorityMatrix*;
else if *Event == device d configuration changed* **then**
  forall $c \in components$ **do**
    forall *liquid-style in the component* **do**
      Check if the *device* accepts the liquid-style and save the highest priority in $priorityMatrix[c][d]$ and in $cloneMatrix[c][d]$;
    end
  end
else if *Event == device connected || Event == device disconnected || Event == user connected || Event == user disconnected* **then**
  forall $c \in components$ **do**
    forall $d \in devices$ **do**
      forall *liquid-style in the component* **do**
        Check if the *device* accepts the liquid-style and save the highest priority in $priorityMatrix[c][d]$ and in $cloneMatrix[c][d]$;
      end
    end
  end
Result: updated $priorityMatrix$ and $cloneMatrix$

Algorithm 6.3.1: Incremental constraint-checking and priority computation

component does not affect the acceptance of the liquid media queries of any other components. When a new component is created (or removed), a row is added

(or removed) to the *priorityMatrix* and the algorithm recomputes the highest priority scores. If a created component defines a liquid media query with the clone attribute, then the highest priority value between the clone styles is also stored in the *cloneMatrix*.

• **The meta-configuration of a device is changed** - when the device *type*, *ownership*, and *role* change, the priority values of the corresponding column are updated for both matrices.

• **A device joins or leaves the current session** - these events affect the *devices*, *min-devices*, and *max-devices* features of the liquid media queries, which triggers the recomputation of the whole *priorityMatrix* and *cloneMatrix*.

• **A user connects or disconnects from the application**; Changes to the *users*, *min-users*, and *max-users* features also require a complete recomputation of the *priorityMatrix* and *cloneMatrix*.

6.3.2 Phase 2: Migration and Cloning

The migration and cloning phase uses the previously computed *priorityMatrix* and *cloneMatrix* to determine on which devices each component should be migrated or cloned on. The algorithm prepares a migration plan where each component is assigned to a given target device. The choice follows the *minimum number of components per device* policy so that the number of components running on each device is minimized, making it possible to spread the liquid Web application across as many devices as possible. If the component instances outnumber the available devices, some of the components will be co-located on the same device. Equation 6.3 shows the resulting *migrationPlan* computed under the constraints of the liquid media queries of the scenario depicted in Figure 9.11 (the constraints are presented in Section 9.3). c_{video} is migrated to d_{tv} with the highest priority, $c_{comments}$ is migrated to d_{laptop} with the lowest. Once the *migrationPlan* is ready, the liquid application can redeploy the components across the set of devices accordingly.

$$\begin{aligned}\text{migrationPlan} = [&\{c_{video}, d_{tv}\}, \{c_{suggestedVideo}, d_{phone1}\},\\ &\{c_{videoController}, d_{tablet}\}, \{c_{comments}, d_{laptop2}\}]\end{aligned} \tag{6.3}$$

After the migration step is complete, the cloning routine can start. This process exploits the *cloneMatrix* computed in phase 1 and the clone rules associated to the components that need to be cloned. All the devices that were not used in the previous migration step are flagged as candidates for running a cloned component. The candidates are grouped and prioritized following the clone rules,

```
Data: priorityMatrix
Data: Shared global state: components, devices
componentOrderedTargets ← {};
migrationPlan ← {};
for component ∈ components do
    componentOrderedTargets[component] ← {};
    highestPriority ← 0;
    targets ← [];
    for device ∈ devices do
        priority ← priorityMatrix[component][device];
        if priority > highestPriority then
            highestPriority ← priority;
            migrationPlan[component] ← device;
        end
        targets.push({device : device, priority : priority});
    end
    orderedTargets ← sort(targets) by priority, decreasing order;
    componentOrderedTargets[component].targets ←
     orderedTargets;
    componentOrderedTargets[component].highestPriority ←
     highestPriority;
end
migrationPlan ← {};
sortedPriority ← sort(componentOrderedTargets) by
 highestPriority, decreasing order;
for component ∈ sortedPriority do
    deviceIndex ← 0;
    do
        unique ← true;
        tempTargetDevice ←
         sortedPriority[component].targets[deviceIndex];
        for d ∈ migrationPlan do
            if tempTargetDevice = migrationPlan[d] then
                unique ← false;
            end
        end
    while unique == true;
    migrationPlan[component] ←
     sortedPriority[component].targets[deviceIndex];
end
Result: migrationPlan
```

Algorithm 6.3.2: The redistribution algorithm for computing the `migrationPlan`. the algorithm encapsulates the *priority-based* and the *minimum number of components per device* policies.

the device that contains the source component that needs to be cloned is never considered as a possible target of the cloning, and every component which can be cloned is associated with the list of target devices. Similarly to the previous step, the algorithm prepares a clone plan that is used by the liquid application for cloning the components. Equation 6.4 shows the output *clonePlan* computed with the matrix shown in Equation 6.2 under the constraints of the liquid media queries of the scenario depicted in Figure 9.11 (see Section 9.3 for the constraints).

$$\text{clonePlan} = [\{c_{videoController}, d_{phone3}\}] \tag{6.4}$$

Algorithm 6.3.2 computes the migration plan by implementing the *priority-based* and *minimum number of components per device* policies. If multiple liquid media queries have the same priority and multiple components can be migrated to the same connected devices, the priority is resolved based on which component was instantiated first in the liquid application. However it is encouraged that the developers give different priorities to their liquid media queries, instead of relying on the time when components are instantiated. While it can be easy to predict when a component is instantiated on a single device environment, it is not trivial to determine beforehand when a component is instantiated if the application is deployed on multiple devices.

The first for-loop in the algorithm orders the priority scores for each component, then, starting from the one with the highest priority, it builds the `migrationPlan`. In this version of the algorithm, the outcome does not consider the overall migration cost in terms of the number of migrations to be performed or the time required to migrate a given component instance. Minimizing such cost would become important when the algorithm is applied to an input configuration of components already instantiated across multiple devices.

6.3.3 Phase 3: Component Adaptation

The *component adaptation* phase happens once the migration and cloning is complete. Each device checks for each instantiated component which liquid media queries are accepted and activates the associated style sheets. The standard CSS mechanisms for dealing with overlapping selectors take over.

6.3.4 Run-time Complexity

The complexity of the algorithm we discussed in Section 6.3 depends on three factors: the number of devices (D), the number of the components (C), and the number *liquid-style* elements (S). In the worst case, the run-time complexity of Algorithm 6.3.1 is $\mathcal{O}(D * C * S)$. However, the actual run-time complexity depends on the event that triggered the incremental version of the algorithm: • $\mathcal{O}(D * S)$ for newly created components; • $\mathcal{O}(C)$ for deleted components; • $\mathcal{O}(C * S)$ for changed device configurations; • $\mathcal{O}(D * C * S)$ for all other events. The run-time complexity of the migration and cloning phase is $\mathcal{O}(C * D^2)$, and the adaptation algorithm explained in Section 6.3.3 has complexity $\mathcal{O}(S)$.

The execution for Algorithm 6.3.1 can be decentralized as the responsibility for computing the priorityMatrix columns can be offloaded on each device, assuming that they all have access to the component liquid style definitions. Each device takes care of updating their columns whenever an event occurs and stores the result in the application shared state, which is automatically synchronized among all devices. We discuss the design of the decentralized algorithm in our prototype in Section 7.5.2.

Part III

Framework Implementation

Chapter 7

Liquid.js for Polymer

The design of the majority of the Web applications does not yet take into consideration the scenario in which the users access their applications with all the Web-enabled devices they possess [Goo12]. Those devices may be used sequentially, where the work made on a device needs to be transferred to another as the users continue to use the application, or simultaneously, where the multiple users collaborate using multople devices. Traditional Web 1.0 applications, which do not use sessions, make it possible to transfer applications across devices by sharing the link to the page currently opened by the users. As soon as sessions are introduced (e.g., when the users need to be authenticated) more effort is required to make the application flow between different Web browsers. Modern Web applications, which keep a significant amount of state on the client, make it even more complex for developers to achieve the LUE as their architecture was not designed to withstand the interactions of the same user using multiple Web browsers on different devices, possibly at the same time.

The goal of the Liquid.js for Polymer framework is to help developers build Web applications that can take full advantage of all devices owned by their users by transparently creating an environment suitable for creating the LUE [132] described in Chapter 3.

7.1 The Framework

The Liquid.js for Polymer framework (referred as just Liquid.js from now on) 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 decentral-

ized Web applications that need to operate on a shared decentralized state that can be deployed across multiple heterogeneous devices. Liquid.js can be used to create level 5 liquid Web applications (as discussed in Section 3.3.4.10).

The idea behind the Liquid.js framework is to give to developers the tools to easily create applications transparently running on multiple heterogeneous devices. As previously discussed in earlier sections, there are three different uses cases of the Liquid.js framework: • **sequential screening**: one user owns two or more different devices, on which the application runs at different times; • **simultaneous screening**: one user owns two or more different devices, on which the application runs at the same time shows different components of the same application deployed on each device); • **collaboration**: two or more users collaborate using the same application running on all of their devices either sequentially or simultaneously.

The assumption is that applications are developed using the Web Components standard [W3C14], which provides the necessary level of granularity to structure the application UI and its state into modular, reusable and composable units that can be independently deployed across multiple devices. While for the sequential usage scenarios it is sufficient to make the whole Web application liquid, a fine-grained component-based approach is particularly suitable for simultaneous usage scenarios. This way, developer may control the deployment configuration of each part of the application or the UI and best decide how to empower the users to rearrange and lay out the Web application across all available devices. Figure 7.1 shows the positioning of the Liquid.js design decisions on the design space of liquid software presented in Section 3.1.

We assume that components are built with the Polymer framework [The18a], developed by Google. Liquid.js is compatible with any of the Polymer components that can be found on the *catalogue of Web components* [Web20] (maintained by the Polymer community) as well as with any Polymer component built by any Web developer that complies with Polymer v1.x rules. Liquid.js transparently takes care of the state synchronization by using Yjs [139], a connector for concurrency control and conflict resolution framework that takes care of synchronizing the state across multiple devices through messages that are then sent over both WebSockets and WebRTC P2P connections.

Developers using Liquid.js need to inject the *liquid behavior* [Pol18] into all the Polymer components they expect to have liquid features, in such a way that they can later be dynamically migrated to run on other devices. E.g., the liquid behavior gives to a stateful Web component the ability to be dynamically deployed, migrated, forked, and cloned on any Web browser-enabled devices importing the Liquid.js API.

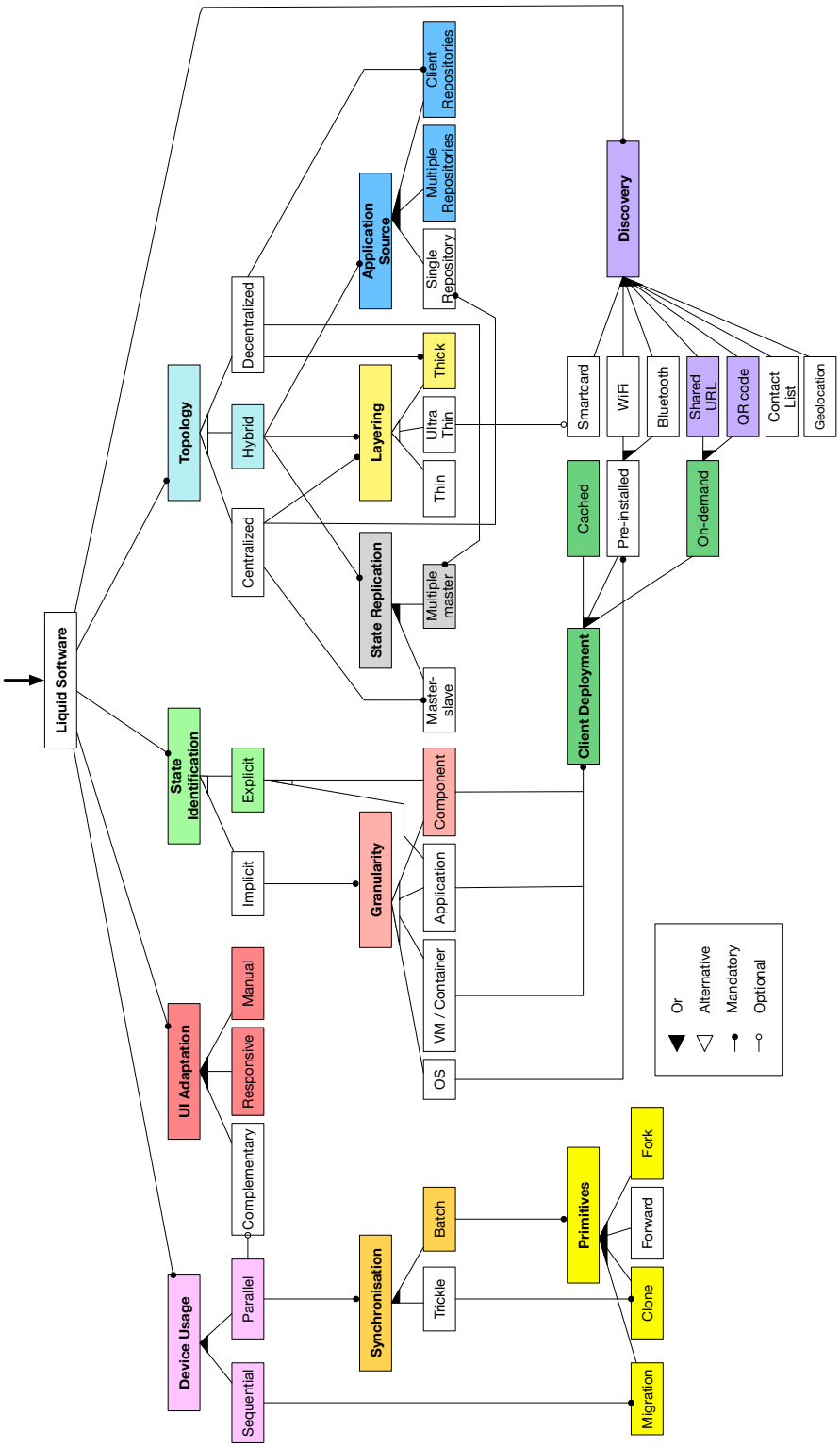


Figure 7.1. Liquid.js design decisions in the design space of liquid software (see Figure 3.1). The Liquid.js features are highlighted with colors.

To do so, the state of a Web component can be annotated following the Liquid.js conventions. Developers can choose which components are liquid and they can explicitly define which properties should be shared with other components upon migration. Liquid.js reads the annotated components and transparently manages the asset deployment, state migration and synchronization across components running on different devices. Polymer components that import the liquid behavior are called *liquid components* and any Polymer property that needs to be synchronized and is annotated as liquid is called a *liquid property*. Liquid properties can strictly be defined only inside liquid components. A liquid component can be instantiated on any *device* running a modern Web browser that is connected to the liquid Web application discovery server.

Liquid.js allows users to instantiate any component provided by the Web application on any of their devices, furthermore it allows users to migrate those components directly across any other device. Whenever a component moves across devices, if the target does not yet own the assets of the component, it will request them from the source so that they can be dynamically loaded on the new device. To do so, Liquid.js supports both a centralized and decentralized approach to distribute and deploy the assets of a Web application. Like any other traditional Web application, the server of the liquid application (see Figure 7.2) stores all the assets of the application (e.g., the HTML, CSS, and JS files containing the definition of liquid components). As assets are downloaded by the clients connected to the application, Liquid.js no longer relies only on the central Web server. Since clients own a copy of the assets they can help the server by sending the assets to new clients connecting to the application. Clients can *distribute* assets to their neighbouring devices through P2P channels created with the WebRTC Peer Connection and DataChannel APIs. Creating a fully distributed architecture from the very beginning is impossible with current Web technologies, because users connecting from their Web browsers do not yet own a public IP address, thus they need to connect to a Web server in any case for discovery purposes. Therefore the server of the liquid application takes care of the discovery of the clients by implementing a *Signaling Server* which can also be used for relaying messages between the devices that cannot create a direct WebRTC connection between them.

Liquid.js internally identifies liquid properties, liquid components and devices with unique Uniform Resource Identifiers (URIs). The framework applies an identifier to each device upon connection and it assigns an identifier to liquid components and their properties whenever they are instantiated. These identifiers can be used as URIs within the framework whenever there is the need to refer to them, e.g., in order to migrate a liquid component from a device to

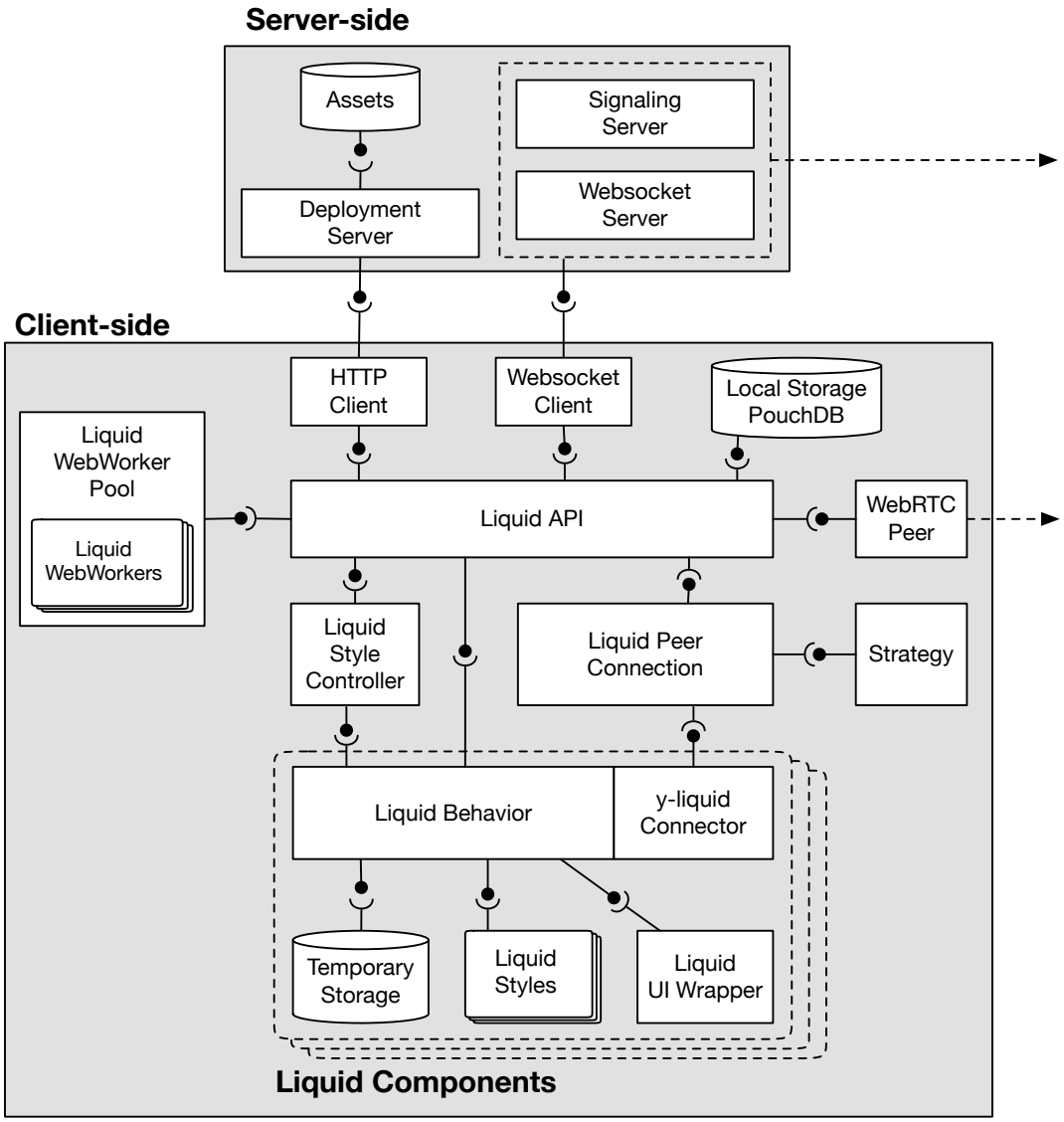


Figure 7.2. Liquid.js simplified architecture

another the source *componentURI* and the target *deviceURI* have to be known. Liquid.js URIs follow the liquid URI scheme and are de-referenceable through the framework and not by using the HTTP protocol. URIs simplify the design of the API as the same methods can be applied both to components deployed on the device issuing the command as well as to remote components.

The liquid behavior transparently communicates directly with the core components of the library, the *liquid API* component, and the *y-liquid connector* component (see Figure 7.2). The latter defines the implementation of a connector for

the Yjs framework [139] which takes care of synchronizing data structures between devices. Whenever the state of the component's liquid properties change, Yjs and the *y-liquid connector* create and send synchronization messages which are automatically delivered to other paired devices.

The existing behavior of any component built without using Liquid.js (legacy Polymer components) is untouched, even if the component is later redesigned to import the liquid behavior. When the component imports the liquid behavior, it still have full access to any W3C HTML5 API or any third party imported library defined in the main JS environment. Liquid.js wraps the solid Polymer components and sets up proxy traps and object observers on the annotated Polymer properties. This approach allows to separate concerns between the liquid behavior and the actual component behavior without requiring developers to change the code they already own. Instead, they only need to explicitly annotate as liquid the properties whose values should be migrated or kept synchronized across devices.

7.1.1 Granularity

The client of Liquid.js is *thick*. Liquid Web applications developed in Liquid.js are *component-based* and all components can be shared between the set of devices. The client is responsible for transparently creating the P2P mesh among the clients; moreover, it hides the complexity of data and state synchronization by hiding the protocol used and by taking care of data consistency among the devices. This approach allows Liquid.js to migrate only small pieces of an application instead of migrating the entire application. By migrating liquid components it is possible to migrate only parts of the UI from a device to another without loading the entire application on all devices.

While the liquid data layer is designed at the component level granularity, Liquid.js also allows to synchronize the state at the property level Section 4.2.

7.1.2 Topology and Code Deployment

The topology of Liquid.js aims to be *distributed*, or when necessary, as *decentralized* as possible. Initially the master copy of the assets of an application reside in a Web server (which can be replicated following the *multiple repository* approach. Whenever clients request pieces of an application in the form of liquid components (*on-demand*), they may decide to *cache* any asset inside the storage of their Web browser. Afterwards any client can request any other to send their own copy of the application. This approach allows Liquid.js to take advantage

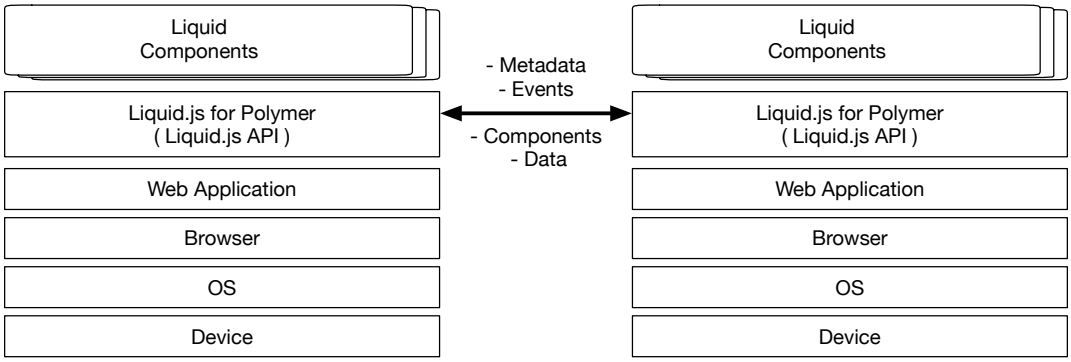


Figure 7.3. Applications built on top of Liquid.js: stack view.

of both *multiple repositories* and *client repository* as the source of the application (as discussed in Section 3.2.1.2). The hybrid application deployment is a mix of the multiple repositories and client repositories topologies shown in Figure 3.3b and 3.3b. The exchange of assets happen through a P2P mesh that is established dynamically at runtime. The P2P channels are created using the WebRTC *RTCDataChannel* API implemented using the Peer.js library [Mic19].

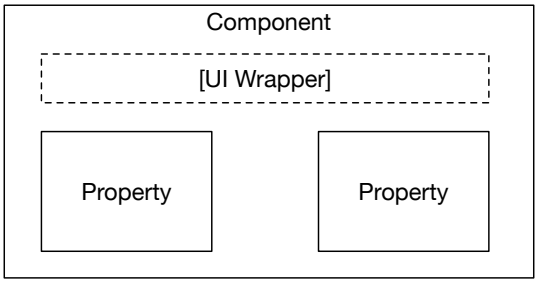
7.2 Liquid Web Applications

Liquid Web applications developed with Liquid.js depend on the Liquid.js API that all liquid applications need to import into their Web page. Liquid.js can run on any modern Web browser that follows the HTML5 standard (e.g., Google Chrome, Mozila Firefox) [Ale19]. Figure 7.3 shows the stack diagram of a liquid Web application built on top of Liquid.js. Liquid components run on top of the Liquid.js API, which acts like a middle-ware in the communication between clients and takes care of sending metadata information, events, components, data and state.

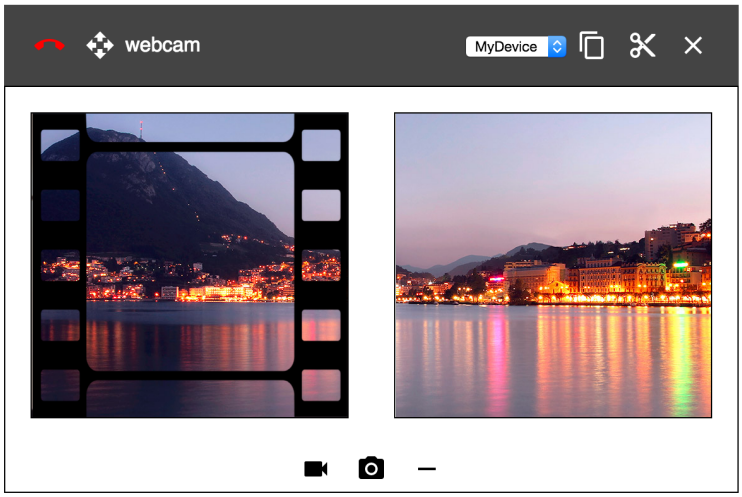
Liquid.js provides developers with the following abstractions (Figure 7.4a):

• **liquid component**: a piece of executable code defined by both JavaScript (JS) and HTML. The liquid component encapsulates both the state of the component and the liquid behavior, which is the core module that provides the Liquid API to the component.

• **liquid property**: the liquid property is the smallest indivisible piece of state contained in a liquid component. The liquid property is defined by a *name*, a *value*, and a set of *annotations*. The state of a liquid component is defined by all liquid properties it contains.



(a) Liquid component structure: a component contains an optional liquid UI wrapper and any number of liquid properties. In the diagram the component has two liquid properties.



(b) Sample liquid component labeled webcam. The top black bar of the component is the liquid UI wrapper, which can be used to control the LUE. In this example we show the `debug` UI wrapper provided by Liquid.js. The component defines two properties: the first one stores the webcam video feed (left); the second stores a snapshot of video (right). The user can take new snapshots by using the three buttons displayed on the bottom of the component.

Figure 7.4. Liquid component structure and comparison with a real liquid component.

• **liquid UI wrapper**: the liquid UI wrapper is an optional UI element that surrounds a liquid component. The wrapper enhanche the LUE and enables the LUE primitives by providing support for the interactions of the users with Liquid.js. The framework provides the `debug` and the `default` UI wrappers that help users to migrate, fork, clone, and delete components. The wrapper can also show to the user the feedback of the available set of connected devices. The de-

velopers of liquid components can choose to use their own liquid UI wrappers, that can be implemented with the liquid API (see Chapter 8).

In Figure 7.4b we show the example of a liquid component created on top of Liquid.js. The *webcam* component takes screenshots from the device's webcam and share them with all other paired instances of the same component. The component defines two liquid properties: • the first contains the video feed of the webcam; • the second stores the screenshot of a frame that can be shared with all other webcam components. The buttons on the bottom of the component are part of the component and can be used to activate/deactivate the webcam, capture a screenshot of the webcam feed, and delete the current taken screenshot. The liquid UI wrapper shown in the example is the `debug` wrapper bundled with Liquid.js, and is displayed as the dark bar above the component. The wrapper include buttons and icons than can enhance the LUE from the perspective of the users of the liquid application, if the developers did not built their own ad-hoc wrapper. The `debug` UI wrapper give access to the following features from left to right: • The phone icon gives feedback on the status of the connection with the signaling server: if the icon is red (e.g., as shown in the figure) the Web browser is currently disconnected with the signaling server, otherwise the icon is green; • The four-arrows icon allows the users to call the *clone* LUE primitive with a drag-and-drop gesture. When the users drag the component, an overlay appears displaying a set of icons which represent the set of connected devices; the users can drop the component on any of the icons and migrate the component on the target device. We discuss a more advanced drag-and-drop UI wrapper later in Section 9.4.4; • The label shows the name of the wrapped component (e.g., *webcam*); • The drop-down menu can be used to select a device in the list of all connected devices, once the device is selected the users can use the next to buttons for accessing the *fork* and *migrate* LUE primitives; • The copy button can be used to *fork* the component on the selected device; • The cut button can be used to *migrate* the component on the selected device; • The close button can be used to delete the component.

7.2.1 Liquid Components

The liquid component is a piece of mobile code [40] which encapsulates a set of *behaviors*, methods and properties (coded in JavaScript (JS)) and the *UI* (written using both HTML and CSS). Whenever a liquid component needs to be instantiated, it is dynamically loaded in the page through the *HTML imports* API [W3C16].

We take a component-based approach because our goal is to have fine-grained

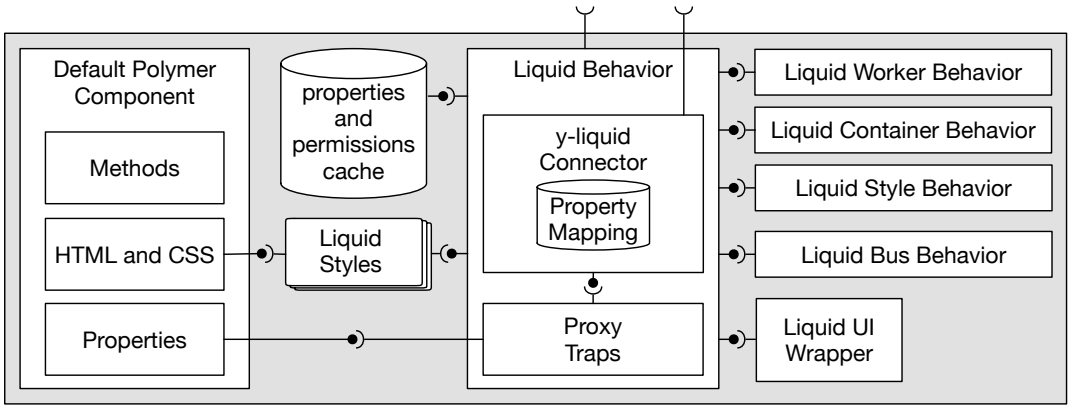


Figure 7.5. Liquid component architecture.

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 the user commands. By decomposing the UI into components we ca also 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. While many Web applications are built following the model-view-controller pattern [110], here 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 across multiple devices.

Liquid components on top of Polymer specifically follow the Polymer syntax [The18a; The18b]. Thanks to the Web components standard the instances of a liquid component are isolated from each other, making it possible to effectively decompose and build applications with multiple instances of the same component, while also making it possible to explicitly describe which part of the component state needs to be synchronized.

Figure 7.5 shows the architecture of a liquid component. A liquid component is defined by two main parts: • the default Polymer component composed by the set of methods, properties and HTML and CSS elements designed and built by the developers of the liquid Web application; • the liquid behavior imported in the Polymer component, which manages the replication and synchronization of its state, the UI wrapper interactions, and the activation and deactivation of the *liquid styles* (the implementation of the liquid styles are discussed in Section 7.5).

Listing 7.1. Liquid.js: how to import the LiquidBehavior into a Polymer component and annotation of a liquid property.

```
1  <dom-module id="liquid-component-example">
2    <template>
3      <!-- HTML here -->
4    </template>
5    <script>
6      Polymer({
7        is: 'liquid-component-example',
8        behaviors: [LiquidBehavior], // Importing the behavior
9        properties: {
10         exampleProperty: {liquid: true} // Property annotation
11       },
12     });
13   </script>
14 </dom-module>
```

Liquid.js takes care of propagating and synchronizing changes of the state for each component instance at the property level across multiple devices. The developer of the component does not need to worry about how many instances are running or keep track of the set of devices that are connected to the application since the framework does it on behalf of the developers. The communication with the other liquid components is also managed transparently from the developer perspective through the *y-liquid connector* built with the Yjs library.

Liquid.js annotations define which components should manifest the LUE and which parts of the state of an instantiated component are meant to be shared among other components. This process is accomplished simply by importing the *LiquidBehavior* class inside the definition of a component and by explicitly defining which properties are liquid (see Listing 7.1). Once the developers add their own annotations, Liquid.js will transparently manage the deployment of the application as well as the state and data synchronization of a liquid component. At line 8 of Listing 7.1 we show how to import the behavior into a component. A component can import any number of behaviors and they are sequentially imported following the array order. Liquid.js defines multiple behaviors other than the *LiquidBehavior* Section 7.2.3, but it should always be loaded before all other behaviors because they are not standalone and always require the core *LiquidBehavior*. A component importing the liquid behavior can be instantiated, migrated, forked, cloned and deleted by the Liquid.js API, however if no liquid property is defined, no state is synchronized across the devices. At line 10 we show how to

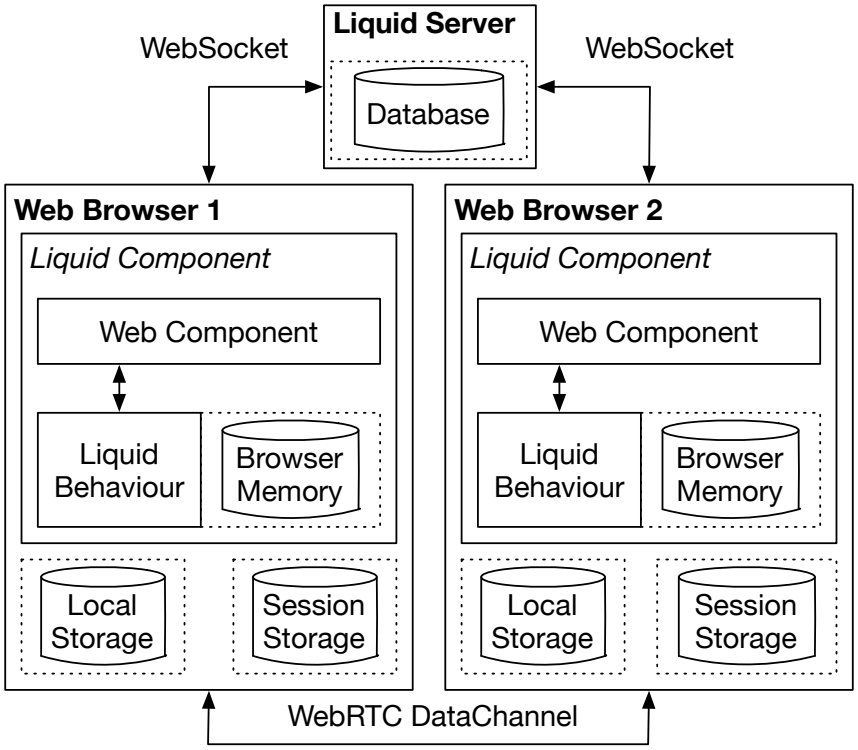


Figure 7.6. Liquid.js runtime and storage deployment built based on the design of level 4 storage deployment shown in Figure 4.7.

annotate a liquid property. Defining a property as $liquid : true$ is enough to define make it liquid and synchronized with all properties it is paired to, however in Section 7.2.2 we present more annotations.

The liquid behavior is able to detect updates to each property through a set of proxy traps created by the liquid behavior for each Polymer property defined as liquid. Proxy traps are created when the component is instantiated with the HTML5 Proxy API [Moz20e] and they can intercept changes to the value of a property and forward them to the *y-liquid connector*, which then propagates the updates to all paired properties.

7.2.2 Liquid Properties

The liquid state of a liquid Web application is decomposed and stored into liquid components and their state is separated into a set of liquid properties. A property is identified by its name and can store the value of any JavaScript Object Notation (JSON)-serializable JS data type.

Listing 7.2. Liquid.js: annotations of a liquid property with the default values.

```
1  properties: {
2      exampleProperty: {
3        type: Object,
4        value: { function(){ return {} } }, // default value
5        liquid: {
6          scope: "local",
7          persistency: "volatile",
8          permissions: {
9            publish: true,
10           subscribe: true
11         }
12       }
13     }
14 }
```

While each liquid component instance always holds the current value of a liquid property, the Liquid.js framework may choose to replicate, store and manage the liquid state outside of the liquid component as we discussed in Section 4.4. In Figure 7.6 we show the storage deployment of Liquid.js. Values of liquid properties are automatically synchronized among paired component instances, according to the permissions associated to each property.

Listing 7.2 shows all possible annotations of a liquid property. The behavior of the annotations follows the description discussed in Chapter 4. The annotations can have the following values:

- **scope** - line 6 - default: "local" - possible values: "local", "shared", "global";
- **persistency** - line 7 - default: "volatile" - possible values: "volatile", "session", "persistent";
- **publish** - line 9 - default: true - possible values: false, true;
- **subscribe** - line 10 - default: true - possible values: false, true;

It is not necessary to define all annotations, in fact any missing annotation or set to the *undefined* value are automatically set to the default value. If the **permissions** annotation (line 8) is missing or *undefined*, then both subscribe and publish annotations are set to true. In the example shown in Section 7.2.2 we showed that a liquid property can be defined as $liquid : true$, in this case the liquid behavior assigns all default annotations to the liquid property.

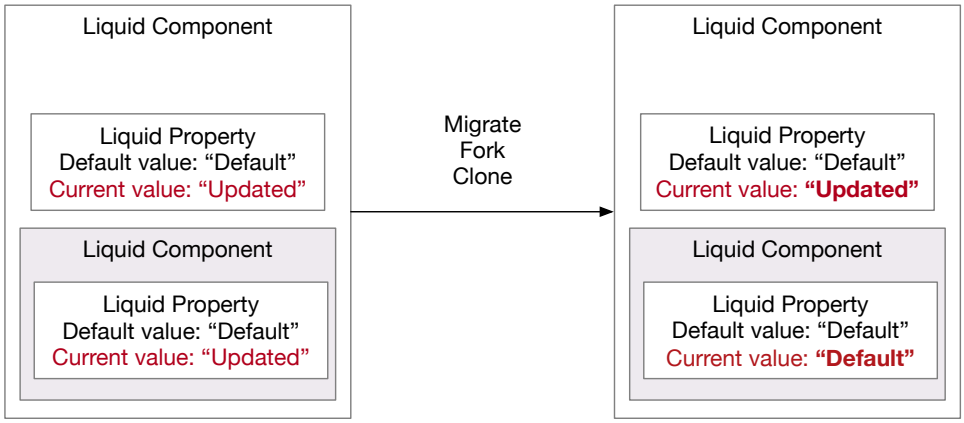
While it is not necessary, it is better to define the *type* of the property as specified by the Polymer syntax (see line 4). The liquid behavior sets up different proxy traps and different callback events inside the *y-liquid connector* depending

on the type of the property. Polymer makes sure that the type of the property does not change at runtime and avoids to change the value of the property if it is not of the defined type. When the type is not defined in the component, Liquid.js tries to infer the type of the property from the default value if it is present (e.g., by using the value defined in the Polymer *value* property as seen in line 4), or by setting it as an *Object* if both type and default value are not specified.

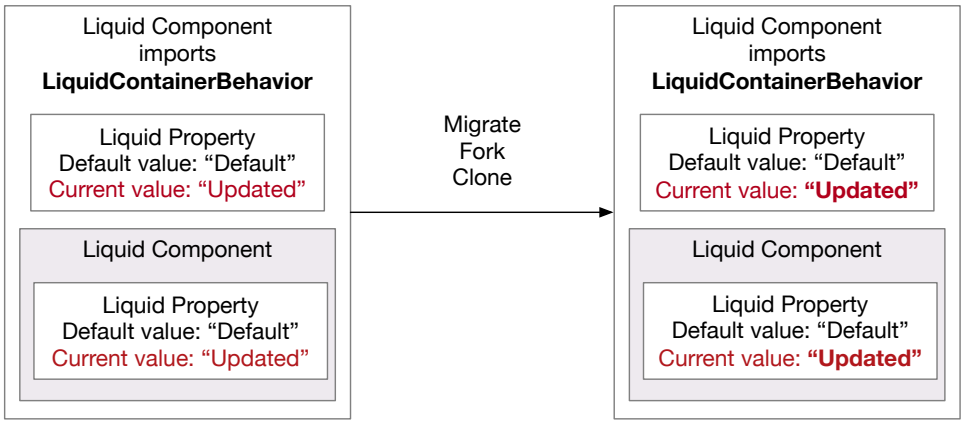
7.2.3 Liquid Behaviors

Liquid.js defines multiple behaviors that can be imported into any Polymer component that add new features to the already existing liquid components. Liquid components can import any number of the liquid behaviors explained below as none exclude the usage of the other. All the behaviors require the core *LiquidBehavior* which must be imported before they are loaded.

- **Liquid Worker Behavior** - the *LiquidWorkerBehavior* allows the developers to bind a LWW to a liquid component (the description of the implementation in Liquid.js of the LWWs is discussed in Section 7.4). LWWs bound to a liquid component are instantiated and paired automatically on any device as soon as the component is instantiated on a device. Components moving to a device that do not have the source of the LWW script also bring a Blob containing the LWW script. This behavior also allow to directly access the LWW without passing through the global LWWPool instantiated in the *Liquid.js API*. This means that the developers can use the LWWs without the need of calling any method on the LWWPool and without the need of instantiating the workers themselves. The *LiquidWorkerBehavior* takes care of the full lifecycle of the LWW. This behavior also allow to use access the *asynchronous data transfer* feature of the LWWs (more about it in Section 7.4.2).
- **Liquid Container Behavior** - the *LiquidContainerBehavior* allows the developers to define liquid components that can contain liquid components. By default the liquid state of a liquid component is defined only by the definition of its own liquid properties, if the component contains an instance of another liquid component, the container does not have access to its liquid properties. This means that when a LUE primitive is called on a liquid component that does not import the *LiquidContainerBehavior* it will not synchronize the state of the contained liquid component. When a component imports the container behavior, then the state of all liquid properties of every component instantiated inside the container are moved together with the container's state. Figure 7.7 shows how the state moves with the container component when the liquid component imports or does not import the container behavior. In both cases the container



(a) The container behavior is not imported: the value of any liquid property of a leaf component is not synchronized when a LUE primitive is called.



(b) The container behavior is imported: the value of any liquid property of a leaf component is synchronized when a LUE primitive is called.

Figure 7.7. Liquid container behavior: behavior of the LUE primitives and liquid data layer with or without importing the LiquidContainerBehavior.

defines a property and contains a liquid component that defines a liquid property of its own. Both liquid properties hold the value "*Updated*" before the LUE primitive is called. After the LUE primitive is called the synchronized state is different: – when the container behavior is not imported (Figure 7.7a), the state of the component inside the container is not synchronized across devices and is set to the default value "*Default*"; – when the container behavior is imported (Figure 7.7b), the state of the component inside the container is synchronized across devices and holds the value "*Updated*".

The container behavior also allows to dynamically create at runtime liquid com-

ponent instances inside the liquid container. Normally the HTML inside a liquid component cannot be the target of LUE primitive (because of the shadow DOM [Moz19l]). The container behavior is not automatically imported in the components contained in a container, therefore all subordinated components must import the container behavior.

• **Liquid Style Behavior** - the *LiquidStyleBehavior* allows the developers to use the liquid media queries defined in the liquid style components as discussed in Chapter 6. The implementation of the liquid style component is discussed later in Section 7.5. The style behavior automatically searches for all liquid style components defined in the liquid component and registers. The behavior is in charge of activating or deactivating the CSS inside of the component when needed.

• **Liquid Bus Behavior** - the *LiquidBusBehavior* allows the developers to pair properties with devices that cannot access the liquid Web application through a Web browser and thus do not have access to the Liquid.js API and the WebRTC protocols (e.g, lightweight Web-enabled IoT devices that can connect to the signaling server, but cannot communicate to the devices directly). When the behavior is imported the developers can annotate their liquid properties with an additional annotation $iot : true$. Whenever the annotation is detected on a property, the behavior eavesdrops the *proxy traps* and propagates the changes directly to the signaling server which then relay the changes to the correct IoT device. Similarly the behavior can also receive updates from the IoT devices relied through the signaling server and update the liquid properties [52].

7.2.4 Liquid UI Wrapper

Developers can create their own liquid UI wrappers by creating a separated Polymer component. The UI wrapper does not import the *LiquidBehavior* as a normal liquid component, but instead imports the *LiquidUIBehavior*, since the wrapper is not a liquid component itself. The behavior transparently registers itself to the liquid behavior of the wrapped component as soon as it is loaded and allows the developers to access the liquid component from within the UI wrapper component by using the keyword `this` [Moz20f] in the component's script.

7.2.5 Uniform Resource Identifiers (URIs)

The design of Liquid.js is based on a hierarchical scheme of three resources: • devices; • components contained in a device; • properties contained in a component. As the resources follow a hierarchical scheme we defined a unambiguous

Listing 7.3. Liquid.js: URIs represented as a JSON objects.

```
let propertyURI = {
    device: ":device",
    component: ":component",
    property: ":property"
}
```

naming scheme to identify them in the scope of the liquid Web application. Liquid.js identifies resources by using Uniform Resource Identifiers (URIs) that can be used anywhere in the Liquid.js API (more about the Liquid.js API in Chapter 8). The URIs follow the naming scheme defined in Equation 7.1. Anywhere in the code of a liquid Web application when a URI is required, the developers can decide to pass the URI targeting a resource in three different ways:

- as a **String** - the String must be formatted following the syntax defined in Equation 7.1 (e.g., `"/:device/:component/:property"`);
- as a **JSON object** - any URI can be represented as JSON object as shown in Listing 7.3;
- as a **direct reference to the liquid component object** (only for component URIs, not available for devices and properties) - if the developers stored the direct reference to a component, they can decide to use the reference instead of passing a URI.

$$/:device/:component/:property \tag{7.1}$$

Developers are allowed to use **wildcards** (*) whenever they write a URI pointing to a liquid property. E.g., in Equation 7.2 we show the URI that resolves as *all liquid properties named "text" contained in all components instantiated on any device*.

$$/*/*/text \tag{7.2}$$

Moreover developers are allowed to write **[:componentTypeNames]** (surrounded by brackets) whenever they write a liquid property URI. E.g., in Equation 7.3 we show the URI that resolves as *all liquid properties named "image" inside the liquid components of type 'webcam' instantiated on any device*.

$$/*/[webcam]/image \tag{7.3}$$

Following this approach it is possible to point to liquid properties deployed across multiple devices. E.g., in Equation 7.4 we show how to invoke the API

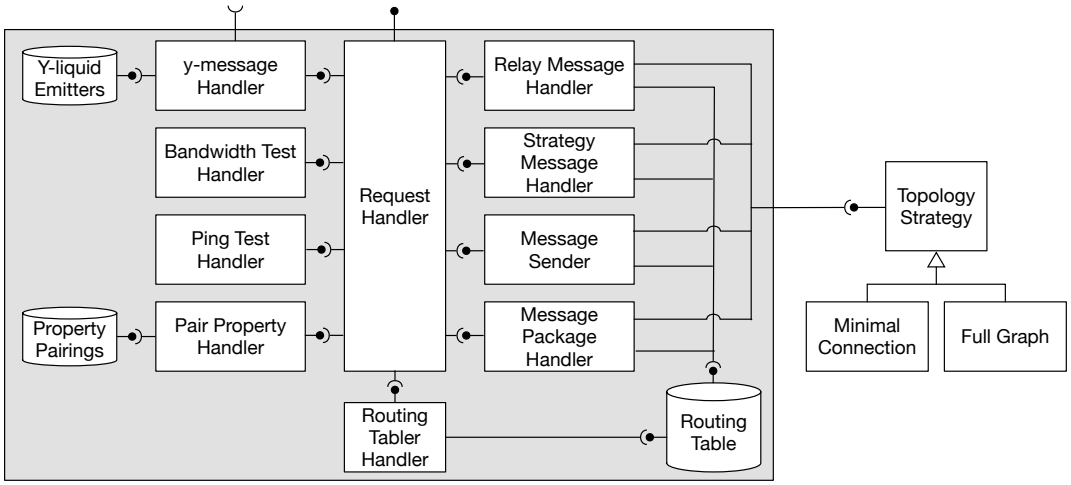


Figure 7.8. Component view of the LiquidPeerConnection (LPC) component.

method *pairProperty* passing two liquid properties URIs. In the example the liquid property named "*image*" contained in component "*c1*" instantiated in device "*d1*", is paired with all other registered *image* properties in the liquid Web application (even if they are deployed on another device).

$$pairProperty('/d1/c1/image', '/*/*/image') \tag{7.4}$$

7.3 Data Layer - Synchronization

Liquid.js is designed to transparently create and dynamically manage the topology network of all the connected peers in a distributed environment. The *LiquidPeerConnection (LPC)* component (see Figure 7.8) [123] manages all incoming messages, both from the signaling server and the peers, and the outgoing messages by deciding which communication channel should be used (e.g., WebSockets or WebRTC). The LiquidPeerConnection (LPC) can create different kinds of topologies depending on a *topology strategy*. Developers can create their ad-hoc strategies by implementing a new class following the provided strategy interface definition (see Section 7.3.1), or they can use the two default strategies provided by the Liquid.js framework: • full-graph strategy; • minimal connections (spanning tree). Throughout this section we explain the design and features of the LPC and how it manage the liquid data layer of the applications built on top of Liquid.js.

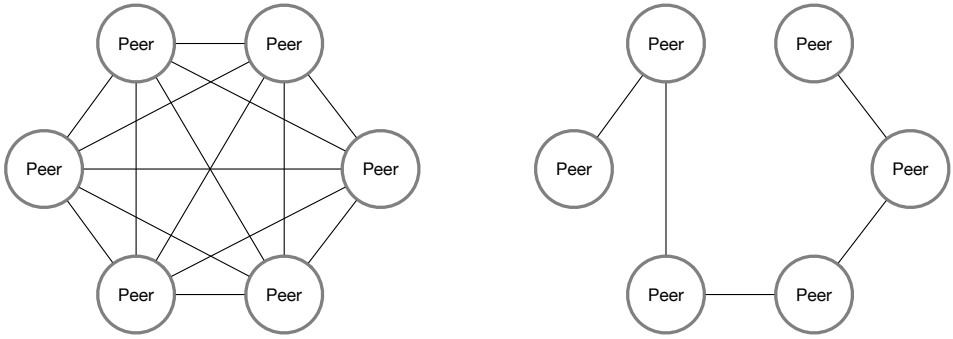
7.3.1 Strategies

The strategies can be used by the LPC to create different peer topologies. The developers can chose their own strategies by changing the appropriate configuration option in the signaling server of the application. The strategies cannot be changed at runtime and all LPC components in all connected peers must adhere to the same strategy. LPC strategies are decentralized and do not store any kind of data on the signaling server. LPC components deployed on multiple peers can communicate with special messages that synchronize their internal state containing their local knowledge of the topology among each other independently from the deployment of the liquid application and without the users' awareness.

Strategies can be employed for overcoming different limits of the devices composing the mesh, e.g., limited bandwidth or limited storage. As the number of devices connected to the liquid Web application increases, so does the number of messages exchanged between peers for synchronizing the state among all of them. For large number of devices, creating a P2P mesh that broadcasts heavy synchronization messages can become unsustainable if the devices cannot keep with the growth in the bandwidth consumption. Peers with a low bandwidth cannot keep the pace of the synchronization and would slowly decrease the responsiveness of the whole LUE. For this reason we created two default strategies and created a strategy interface that can be used to create custom ad-hoc topologies.

Depending on the quality and stability of the network, the developers can create strategies that take proactive routing decisions, e.g., the routing of the Optimized Link State Routing (OLSR) protocol [1] can be used in stable networks in which peers seldom disconnect to pre-compute the best path connecting all pairs of peers. However, for more volatile networks with unstable peers, the OLSR protocol may become inefficient, since the peers' disconnection would trigger the protocol to recompute all best paths The reactive protocol Better Approach To Mobile Adhoc Networking (B.A.T.M.A.N.) [96] can be used for networks that can have a fast evolution, since every peer does not have the knowledge of the whole topology, but only knows about the next best neighbours to reach a given target peer. When a peer disconnects, the B.A.T.M.A.N. protocol does not need to constantly update the state of the topology inside all other peers, which reduces the overhead as compared to the OLSR approach.

The devices connected to a Liquid.js application usually create a volatile environment in which devices do not always have a stable connectivity due to their mobile nature. For this reason two default strategies are implemented in Liquid.js: a reactive and a proactive strategy. Developers can choose which one to



(a) Full graph (b) Minimal connections

Figure 7.9. Default Liquid.js peer topologies example.

use in the configuration file of the Web application. The strategies developed in Liquid.js also have access to features that can be found in a Multipoint Control Unit (MCU) [189], e.g., *message relaying* through another peer or *message packaging*. The advantage of MCUs is that they can coordinate the distribution of the messages among the peers with a single outgoing connection to another peer, instead of connecting to the each connected device. Creating a single connection, instead of a full-graph mesh, reduces the overall bandwidth used by the device when it needs to broadcast messages to all other peers.

Moreover by creating a strategy interface, we also allow the developers to create new strategies based on more recent and advanced technologies once they become powerful and stable enough to be used. E.g., the WebBluetooth API [Web17] can be used to create topologies that depend on the relative position of the devices, and optimize the connections by creating channels between devices that are physically close to each other.

The two default strategies provided by Liquid.js are:

• **Full graph** strategy Figure 7.9a - this strategy creates a full mesh of peers interconnected with each other. The LPC component reacts as soon as a new peer connects and takes care of creating the appropriate communication channel with it (e.g., WebSockets or WebRTC). Messages exchanged through the peers in the full graph strategy are always sent directly to the target, without relaying messages through any other peer.

• **Minimum connection** strategy Figure 7.9b - as opposed to the full graph, this strategy minimizes the total number of communication channels among the peers by building a spanning tree of the full graph mesh. Whenever a new device connects to the liquid application, the LPC does not destroy any of the existing communication channels and create a direct communication channel connecting

it to the peer that detects the connection first, in such a way that the new peer is available as soon as possible in the topology. Messages exchanged among peers in this strategy sometimes relay through the other connected peers.

Custom strategies implemented in Liquid.js must define the following three methods of the strategy interface:

• **choosePath(destinationPeerURI)** - this method computes the next *hop* to reach a destination. A hop can either be the actual destination peer that needs to receive a message, or it can be another peer that is used to relay the message towards the destination. This is the method where the developers can chose to re-actively or a pro-actively route the message towards the target peer, since in here they must decide which path the message takes to reach the final destination. E.g., the full-graph strategy always returns the target destination, because the strategy creates a direct connection between every single peer. The minimal connection strategy can either broadcast the message to all peers it is connected to (and eventually the message will arrive to the destination), or more efficiently it can exploit the knowledge stored in the *routing table* in the LPC to return the next peer towards the destination. The implementation of the routing table is discussed in Section 7.3.2.3.

• **incomingMessage(message)** - this method can be used to receive special messages sent from other LPC components. The purpose of the message is for coordinating the local state of the strategy and can have any payload in JSON format. The strategy can send messages to other LPC components deployed on a remote device by sending a special message of type *type*="*StrategyMessage*" by calling the method *sendMessage* accessible from within the scope of the strategy. This special type of message is recognized by the LPC and it is automatically forwarded in the strategy through this method. E.g., this method is used for by the minimal connection strategy in order to discover any pre-existing path to reach a given destination.

• **ondisconnect(peerURI)** - this method is triggered when the LPC receive a disconnection notification. This can help the strategy to dynamically adjust the internal state of the topology. E.g., this event is not used in the full-graph strategy, because once a peer it is disconencted it will never be the destination of any further message. In the minimal connection strategy, *ondisconnect* allows to trigger a reconnect procedure in order to preserve the current topology state. The strategy attempts to reconnect to the missing peer until a new call of *choosePath* is invoked, or until a timeout is triggered, in both cases if the peer did not respond, the strategy connects to another peer if the spanning tree topology is broken.

7.3.2 Features

The LPC component also implements multiple features that are used by the strategies or can be used by custom strategies for improving the overall topology performance.

7.3.2.1 Ping

The *ping test* component can be used to computing the ping between peers. The strategies can invoke the ping test at anytime and the ping can be affected by the load of the network, as the messages can be queued by the relaying peers. Custom strategies are encouraged to avoid running multiple ping test concurrently, since they could congestion the network and thus return skewed results. The *ping* values are computed in $[milliseconds]$ and are computed by measuring the time it takes for a special custom message to reach a destination peer and backwards (Round Trip Time (RTT)).

7.3.2.2 Bandwidth

The *bandwidth test* component can be used to compute two values: the upload and download speed. The bandwidth test measures the span of time required for a message with a payload of 10MB to reach a peer and then measure the time for the message to come back to the initial peer. The test then divides the size of the message by the elapsed time it took to transmit the payload through the connection channel. The times are approximated to the millisecond and are computed with the internal clocks of each peer. As a result, the clock drift between the different peers and the results may not be symmetric. As this dissertation is being written, the WebRTC Statistics draft [Moz19m] is being updated and the API [Moz19m] written and could be used handle these measurements natively without the need of the *bandwidth test* component.

The *bandwidth test* measurements implementation is similar to the ping measurement, but the messages have a specific payload size of 10MB which allow to more precisely measure the bandwidth speed. The size of the ping message is so small that it cannot be used for measure the bandwidth. The tests returns a value representing the speed with $[megabytes/second]$ unit.

7.3.2.3 Routing Table (RT)

The *Routing Table (RT)* is a feature that can be enabled or disabled in the LPC configuration (see Section 7.3.3). When it is enabled, the *RT* component trans-

parently manage and create a table stored inside the LPC which contains the information about the next hop needed for reaching all peers in the topology. The RT is formatted as a JS $Map()$, where the key represents the destination peer URI and the value represents the next hop URI. Whenever a peer connects or disconnects, the RT component broadcasts a special lightweight probe message asking its neighbours how to reach the others peers in the topology. Once all probing messages receive an answer, the local RT is updated and can be used by the strategies.

The full-graph strategy does not benefits from the RT, which actually creates unnecessary probe messages between the peers. However, as discussed in Section 7.3.1, the minimum connection strategy can benefit from the RT feature. Instead of creating the probing inside the strategy, we provide this useful feature natively in the LPC, so that custom strategies can use it.

7.3.2.4 Packaging Broadcast Messages

The *packaging broadcast messages* is a feature that can be enabled or disabled in the LPC configuration (see Section 7.3.3). Packaging messages allows the LPC component to send less data when it needs to broadcast a message to all peers in a topology. Instead of sending an individual message to each connected peer, the LPC sends a special *packaged* message only to its neighbours, which then are instructed to forward it to the next hop until everyone received the broadcasted information. The special packaged messages contain and array of destinations that is computed and updated by the RT handler. Enabling the packaging feature automatically enables the RT.

The full-graph strategy does not benefit from the packaging feature, which in turn only increases the size of the messages exchanged by wrapping them in a package that is not forwarded to any other hop. The minimum connection strategy can benefit from this feature, because with the packages it decreases the total number of messages exchanged between peers, making it possible to lower the total bandwidth cost of the operation.

7.3.3 Configuration

The LPC component can be configured by defining the LPC property in the Liquid.js API configuration object. Listing 7.4 shows an example of the LPC configuration flags and their default values.

Listing 7.4. Liquid.js: configuration flags of the LPC.

```
const config = {
  lpc: {
    strategy: 'full_graph', // other possible values: "
        minimum_connection" or "custom_strategy"
    useRoutingTable: false, // other possible values: true
    packageBroadcastMessages: false, //other possible values: true
    custom: {/* ... */} // other custom flags that can be used by
        custom strategies
  }
}
```

7.4 Logic Layer - Liquid WebWorkers

Liquid.js implements the liquid logic layer exploiting the Liquid WebWorker (LWW) described in Chapter 5.

7.4.1 Implementation

Figure 7.10 illustrates a simplified component view of Liquid.js extended with the LWWPool. The LWWPool is managed by the framework itself, hidden behind its own API. The framework manages inter-device communication through the *LiquidPeerConnection* component described in Section 7.3. Developers who wish to use the LWW offload feature from outside the scope of a liquid component can invoke the *callWorker* method exposed by the Liquid.js API. The Liquid.js framework also allows to automatically create workers on other machines whenever the *updatePairedDevice* method is called, which guarantees that a copy of each LWW can be found on all paired devices.

7.4.2 Synchronous vs Asynchronous Data Transfer

In Section 5.2 we described that the messages exchanged between devices also contain the corresponding input data that has to be passed to the LWW in order to complete the task. In our prototype we deploy the LWWPool on top of the Liquid.js framework, which already transparently and automatically synchronize the state of liquid properties between devices.

If the data used inside the LWWPool is stored in a liquid property, then we do not have to send it together with the task offloading message, because the state is already synchronized and available on the target the devices. In Figure 7.11

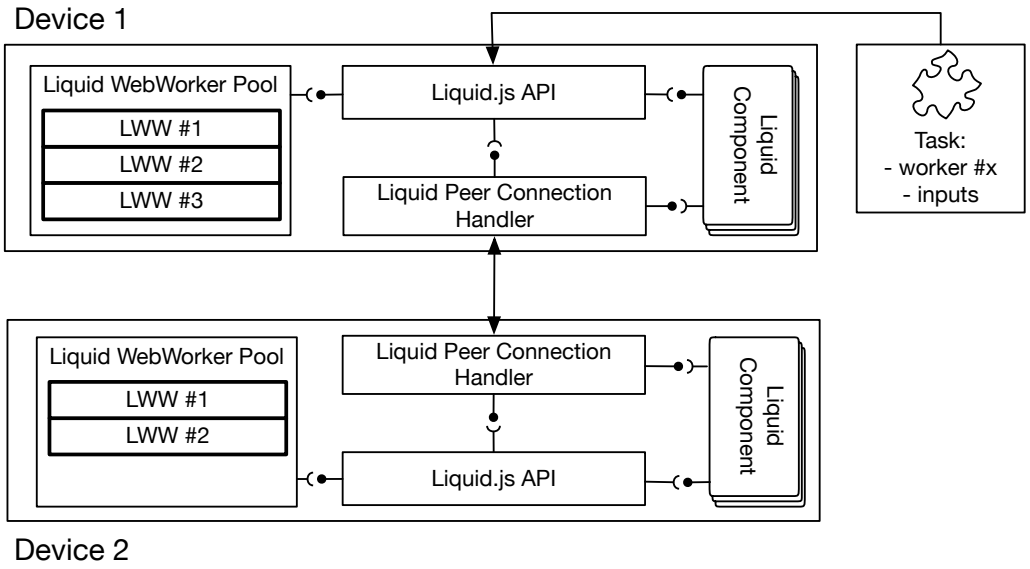


Figure 7.10. Component view of the implementation of liquid WebWorkers inside the Liquid.js for Polymer framework.

we show that we can abstract and separate the flow of data and the task offloading with two different channels. Data is synchronized between all paired liquid components, while tasks offloading messages are exchanged between the LWWPools. Whenever the data should be loaded or saved in liquid property, then the LWWPool is allowed to interact with the liquid components directly. To take advantage of this feature, the developer of the application must call the LWW from inside the liquid component, which transparently will allow the LWWPool to access the data and update it. The task input and result will be automatically synchronized among all paired devices.

In Figure 7.12 we show how we extended the protocol between the two devices with the asynchronous data transfer. In the synchronous version discussed in Chapter 5 the payload of the message (*msg*) contains all the data needed by the LWW to execute the task, now that we rely on the data synchronization of Liquid.js, *msg** contains only the data that is not stored and synced by Liquid.js. For the rest of the data that has to be consistently synchronized between the devices, we pass the liquid property URIs referencing the location of the input/output data. Since data is synchronized asynchronously with respect to the task offloading, we cannot guarantee that when the remote device receives the task offloading request, it also already holds the latest version of the corresponding

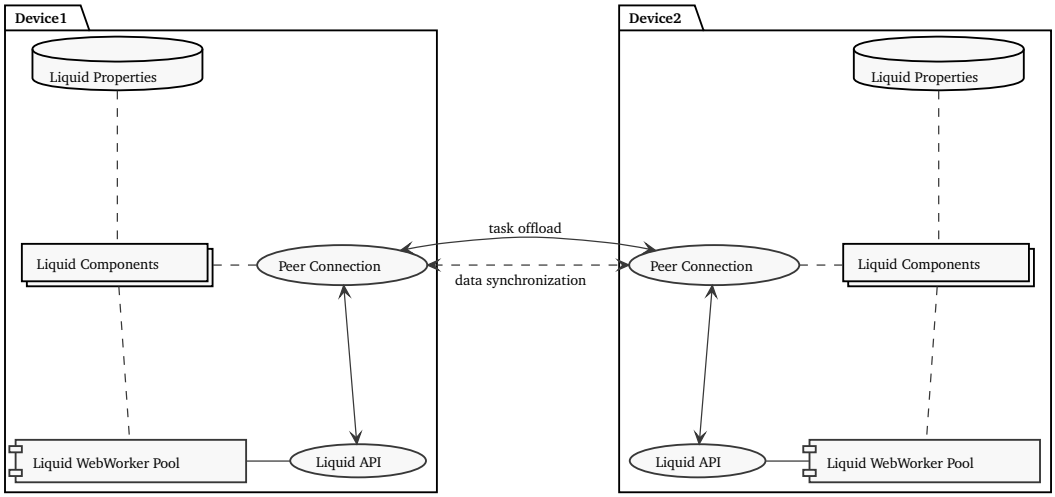


Figure 7.11. Asynchronous data transfer: the dashed lines represent the flow of the data, the full lines represent the flow of task offloading

input data. For this reason, whenever the device offloads a task, it also needs to specify which version of the liquid property the remote device needs to use in order to begin the task execution. If there is at least one *liquidPropertyURI*, then the LWWPool will load the state of the liquid property and pass it to the WebWorker. Once the execution finishes, the remote device immediately notifies the other device that it finished executing the task. The message includes the URI and the new version of the updated liquid property. If any liquid property changed during the execution, these will also be automatically synchronized to make the task execution result accessible across all paired devices. Again, the task completion notification and its output propagation happen asynchronously.

What are the advantages of this approach? In the first place messages exchanged between the devices while performing the task offloading are smaller as they carry a reference to the data vs. the actual data, meaning that communication between the two LWWPools is faster. Additionally, developers can access to two distinct events: *executionEnd* and *dataSynchronized*, these two events can help the developers to report the current status of the application to the user, or they can be used to queue new executions as soon as they are finished. Nevertheless if the data resulting from the offloaded computation has to be sent to the original device, this will require to wait until the liquid properties values have been synchronized. Since Liquid.js data synchronization was developed using the Yjs [139] library, only incremental changes are sent, which results in less data to be sent. More in detail, whenever a JS object property is modified, only the changed property is synchronized, while in the synchronous mode, a copy of

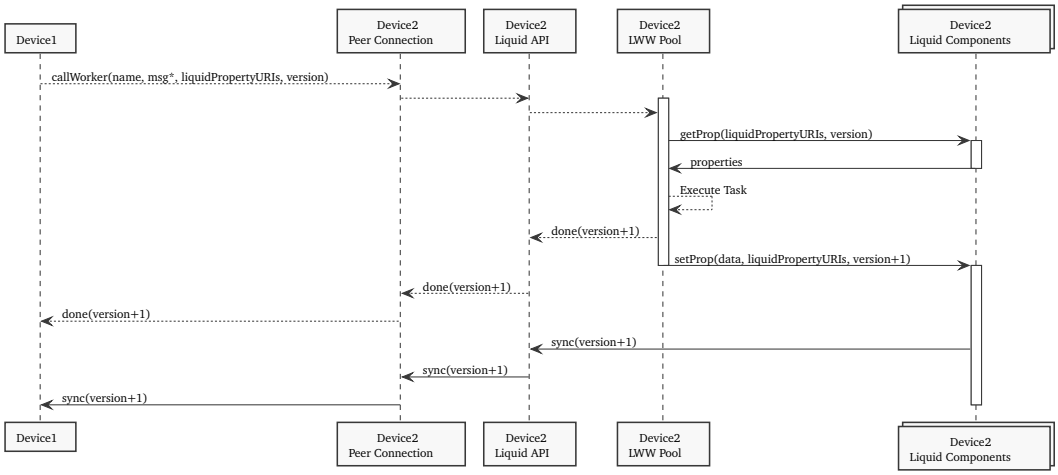


Figure 7.12. Sequence diagram for asynchronous data transfer

the whole object needs to be transferred. Additionally, repeated task executions over the same input data can be offloaded without repeatedly transferring the same data along with each offloading request.

7.5 View Layer - <liquid-style> Component

Standard CSS3 media queries do not allow developers to define new types, features, nor they support customizing existing ones [W3C19] as we discussed them in Chapter 6. The solution we designed for extending the standard media queries is to create a new Web component called *liquid-style*.

The *liquid-style* element shown in Listing 7.5 allows developers to write their own liquid media queries and encapsulate a standard CSS style sheet that is activated when the media query expression is accepted by the device. The *liquid-style* component allows developers to assign values to its attributes (e.g., `device-role`) that can be mapped to the previously defined liquid media types and features by adding the `liquid-` prefix (e.g., `liquid-device-role`). Developers can define their own liquid media queries by assigning values to the corresponding attributes, as shown in Listing 7.6 and 7.7.

In the first example, the liquid media query expression contains both the `liquid` feature and the `liquid-device-type`. Inside the liquid-style component it is not necessary to explicitly set the `liquid` attribute to *true*, since it is the default value for the liquid-style element. The `liquid-device-type` value is mapped to the `device-type` attribute.

The second media query expression contains the liquid media features

Listing 7.5. Liquid-style element and all available attributes.

```
<liquid-style
  liquid                        // Default:"true"
  devices=""                    // Default: ""
  min-devices="" max-devices="" // Default: ""
  users=""                      // Default: ""
  min-users=""  max-users=""    // Default: ""
  device-ownership=""           // Default: ""
  device-role=""                // Default: ""
  device-type=""                // Default: ""
  priority=""                   // Default: "1"
  clone=""                      // Default: ""
  css-media=""                  // Default: ""
> <!-- CSS Stylesheet --> </liquid-style>
```

Listing 7.6. Liquid media query expression mapped to the corresponding liquid-style attributes.

```
@media liquid and (liquid-device-type:phone) {
    body { flex-direction: row; }
}
<!-- Maps to --->
<liquid-style device-type="phone">
    body { flex-direction: row; }
</liquid-style>
```

`liquid-device-role` and `min-liquid-users`, which map directly to the `device-role` and `min-users` attributes. Furthermore the expression also defines the standard CSS3 media feature `min-height`, which in the liquid-style element must be written into the `css-media` attribute.

7.5.1 Design

The automatic complementary view adaptation is achieved through the liquid media query expressions that both define when styles should be enabled on a device and constrain where the components should be migrated if any device with the appropriate features connects to the application. The `liquid-style` component is designed to be attached directly to a `liquid-component` and bundled with a standard Polymer component. Our current implementation of the redistribution process follows both the `priority-based` and the `minimum number of`

Listing 7.7. Liquid media query expression including standard CSS media features mapped to the corresponding liquid-style attributes.

```
@media liquid and
       (liquid-device-role:controller) and
       (min-liquid-users:3) and
       (min-height:900px) {
    :root { background-color: red; }
}
<!-- Maps to --->
<liquid-style device-role="controller"
              min-users="3"
              css-media="min-height:900px">
    :root { background-color: red; }
</liquid-style>
```

`components per device` policies (as discussed in Section 6.2.1).

Liquid.js implements the two LUE primitives that are needed for the redistribution and cloning of the components. The `migrate` primitive allows components to be moved from a source device to another, while the `clone` primitive can be used to copy components and keep them synchronized across multiple devices. Furthermore Liquid.js transparently and automatically creates a synchronized shared state between all connected devices. The shared state contains all the information about the current deployment configuration, such as the number of users connected and the information linked to their set of devices, such as number, ownership, type, and role. The devices can synchronize this information by sending direct messages in a P2P mesh without requiring to relay messages through a Web server.

Figure 7.13 shows how the liquid components are built on top of the liquid application, meaning that each component has access to the Liquid.js API and therefore has direct access to the LUE primitives migrate and clone. Each liquid component can define multiple liquid styles and the framework automatically extracts the liquid media query expressions from within every instantiated component and shares them with all other connected devices, so that each device can check whether it would satisfy the liquid media queries or not. Whenever a query is accepted on a device, that device becomes a possible target for the migration of the corresponding component. When multiple devices become a possible target for the same liquid component, Liquid.js selects the target following the priority-based policy.

Since all the information of the connected devices is stored in the shared state

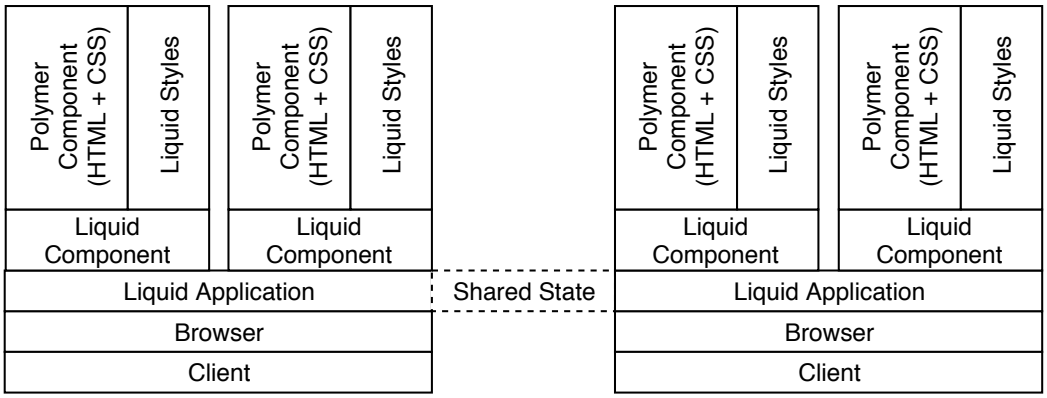


Figure 7.13. Liquid.js: the liquid style elements are bundled with the Polymer component inside each liquid component. The Liquid.js API, represented in the diagram as the liquid application, maintains and keeps the shared state up to date.

of the clients, each device is able to compute new deployments and perform the migration and cloning of the components.

7.5.2 Decentralized Algorithm

The algorithm we proposed in Section 6.3 can run on a Web server, however our goal is to keep the computations of the liquid application closer to the devices of the users. The reason for our choice is twofold: 1. we allow the liquid application to be adaptive even if the Web server goes offline; 2. we enhance privacy because it is not necessary to store on the Web server any information about the users' devices.

In Figure 7.14 we show the architecture we designed to decentralize the *re-distribution and cloning* algorithm. We introduce two new components:

- **`liquid-style` controller**: the controller is in charge to observe any change in the shared state. It interacts directly with the framework API and monitors all events occurring in the set of connected devices (e.g., it monitors for new connected devices). When an event is triggered, it propagates the event to all liquid components that load the `liquid-style` behavior.
- **`liquid-style` behavior**: the behavior gathers information from all instantiated `liquid-styles` and broadcasts messages received from the controller to the `liquid-styles`. New `liquid-styles` register to the behavior as soon as they are instantiated. The instantiated `liquid-styles` can enable and disable styles properly only if the behavior is loaded inside the liquid component.

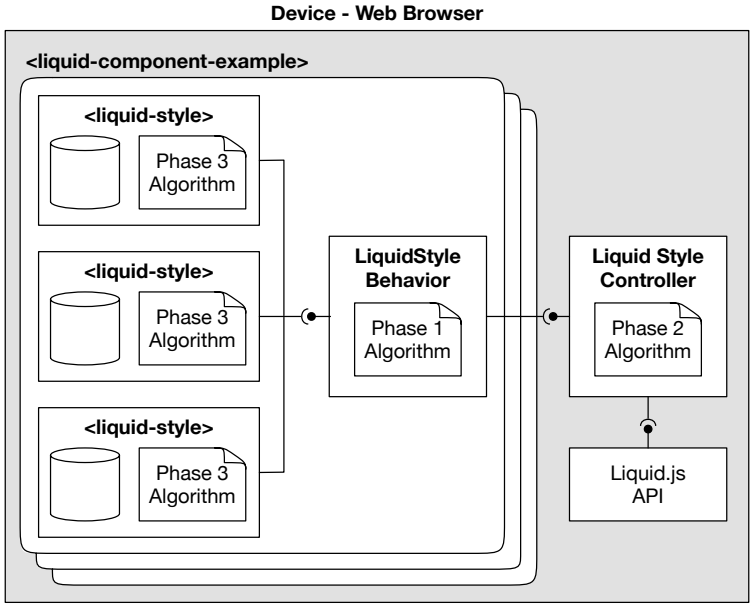


Figure 7.14. Component view of the `liquid-style` component and how it is connected to the `liquid-style` behavior and controller. The phase 1 algorithm (see Section 6.3.1) is encapsulated inside the `liquid-style` behavior; the phase 2 algorithm (see Section 6.3.2) is encapsulated inside the `liquid-style` controller, and the `liquid-style` component is in charge of running the phase 3 algorithm (see Section 6.3.3).

In Figure 7.15 we show how the controller, the behavior and the `liquid-styles` interact during initialization. Immediately after the liquid component is loaded, the behavior starts running and awaits for the registration of new `liquid-styles`. Once all the liquid-styles are loaded, the behavior notifies the controller that the liquid component is ready. The controller immediately creates a new row in the `priorityMatrix` and `cloneMatrix` in the shared state, and then subscribes to it. Liquid.js automatically and transparently synchronizes the state without blocking any device. After the subscription, the controller retrieves the last version of the deployment configuration and pushes it into the behavior. The behavior notifies all `liquid-styles` which then will check if there is a match between the liquid media queries and the current deployment configuration. If there is a match, the style it encapsulates is enabled, otherwise it is disabled. Once all components in a liquid Web application are loaded, it is possible to compute the redistribution and cloning of the deployment.

The decentralized execution of the algorithm is shown in Figure 7.16 and is initially triggered by the actions of the user, e.g., the user connects with

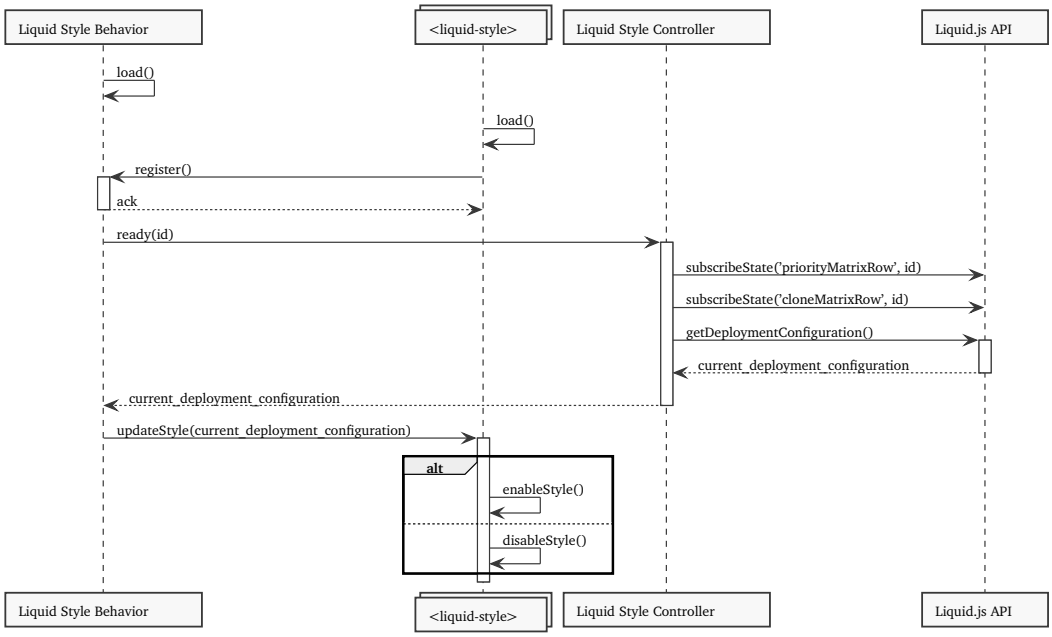


Figure 7.15. Sequence diagram of the initialization of the `liquid-styles`, `liquid-style` behavior, and `liquid-style` controller.

a new device, or changes a device role. When the deployment configuration is changed, Liquid.js catches the event and updates the shared state between the devices accordingly. Once the synchronization finishes all connected devices react and propagate the new deployment configuration to the `liquid-style` controllers. The controllers then send the new configuration to all `liquid-style` behaviors which previously registered to them. The behaviors recompute the `priorityMatrix` and `cloneMatrix`. The phase 1 algorithm described in Section 6.3.1 is ran by the behavior, but instead of computing the whole `priorityMatrix` and `cloneMatrix` as we previously presented, the behavior computes only the rows associated to their own liquid component. The rows of the two matrices are then sent back to the controller which takes care of updating the shared state.

Phase 2 starts when all the rows in the matrices are updated. In order to prevent that multiple devices redistribute the same components multiple times, we need to run the algorithm described in Section 6.3.2 one single time. The most powerful device is selected by the Liquid.js framework [71] and it computes both the `migrationPlan` and the `clonePlan` inside the controller component. The controller then starts the redistribution and cloning phase by calling the corresponding LUE primitives in the Liquid.js API.

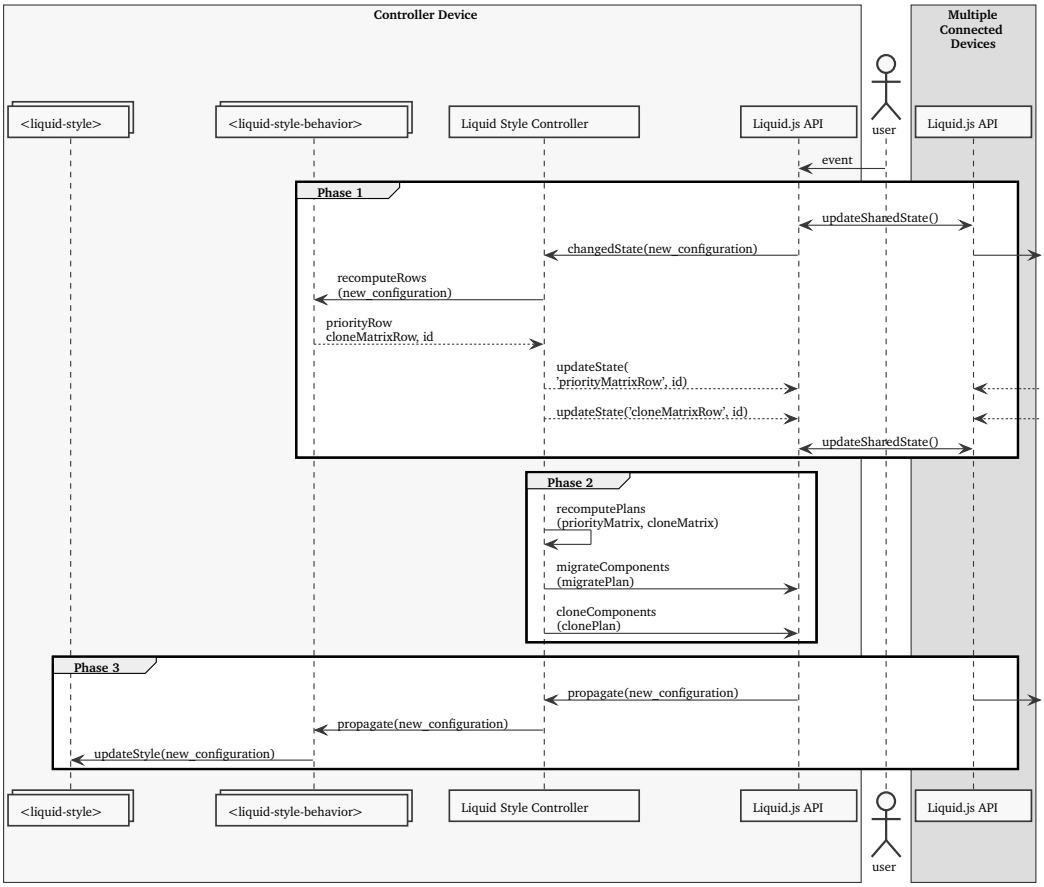


Figure 7.16. Decentralized algorithm processing

Finally phase 3 starts when the components are migrated and cloned. The API propagates to the controller all events triggered by the migration, which are then furthermore propagated to the behaviors. The behaviors broadcast the new deployment configuration to all `liquid-styles`, which then run the phase 3 algorithm described in Section 6.3.3.

7.5.3 Impact

In this section we discuss the impact of the liquid media queries on the design of liquid Web applications. In this section we designed the multi-device adaptation targeting the needs of the developers, whose goal is to create software that can take advantage of multiple devices with the goal of increasing the overall usability of the application. Ultimately, however, the effect of the liquid media queries will be experienced by the user that interacts with the liquid Web application.

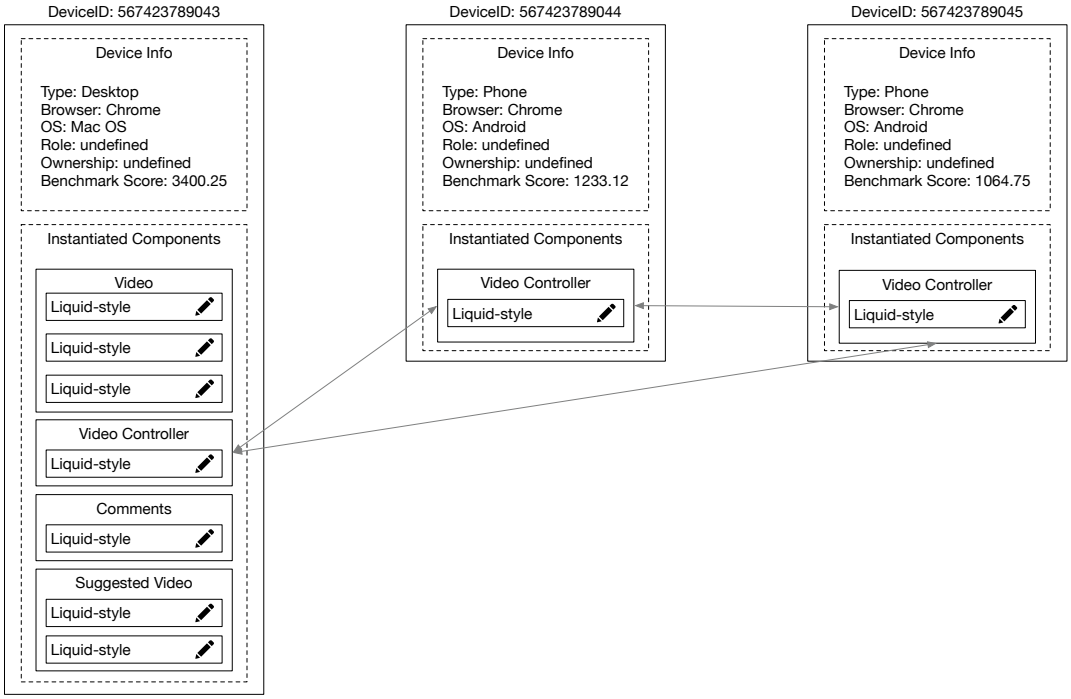


Figure 7.17. Liquid media query debugger tool view.

7.5.3.1 Developers

The multi-device adaptation introduces a new level of complexity that the developers have to face during the design process of their applications. When the developers decide to shift from a single device deployment to a multi-device one, they do not only have to deal with the responsive design of the application, but they are also required to determine how and when components must be migrated across different set of devices. The decision of performing a migration can be driven by technological constraints (e.g., a component requires a sensor that only found on some devices), and/or driven by social interactions [102] and context [145]. Taking into consideration these aspects can be difficult for the developers, and in some cases it can be hard to predict the impact of their multi-device adaptation on all possible sets of users devices. In fact the number of possible combinations of devices owned by the users can grow very fast.

Since so many new facets need to be taken into consideration during the design process, developers would greatly benefit from testing and debugging tools [107], helping to simulate the deployment and observe the behavior of liquid Web applications in a virtual multi-device environment.

In Figure 7.17 we show a view of our debugger tool for liquid styles. By using

the tool the developers can monitor at runtime the evolution of the deployment information stored in the shared state of the application. The debugger visualizes all information about the connected devices, their meta-information (e.g., identifiers, types, roles, ...), the instantiated components deployed on the devices and their instantiated `liquid-styles`. Moreover the tool shows which components are cloned across the devices by connecting two components with an arrow. Since the view is updated in realtime, any time the set of devices is affected by a new event, the view updates and displays the new deployment after the redistribution and cloning process finishes executing. Currently the developers can also read and directly edit the style-sheet encapsulated inside the `liquid-style` components.

In the future we plan to add the following features:

• **Edit liquid media queries**: the developers can edit the media queries at run time (e.g., change the value of a liquid media feature or type).

• **Add or remove `liquid-styles` at runtime**: even if it is already possible to add or remove `liquid-style` components at runtime thanks to the design of the `liquid-style` behavior, the UI of the debugger tool does not yet allow developers to create new styles on the fly.

• **Edit device metadata**: the developers can alter the metadata associated to the connected devices (e.g., type, role). Currently the developers can alter the metadata only from within the corresponding device, however altering the data in the debugger tool is faster and will immediately give back a feedback to the developers about the overall state of the deployment.

• **Add and remove virtual devices**: the developers can create virtual devices and connect them to the application even if they do not physically own them. This feature would allow the developers to simulate deployments that otherwise they could not test.

7.5.3.2 Users

From the perspective of the users, the concerns are different. The users care about their own satisfaction and engagement while they use the displayed UI, and generally they can disagree with the automatic adaptation rules set by developers. In our current approach, the developers are in full control of the liquid media queries and the algorithm does not consider the user needs and opinions when it computes the redistribution.

Still, we believe that the users should remain in control of the deployment of the application on their own devices, and that they should be able to override any decision taken by the algorithm at any time. This can be done in many forms:

• **Edit liquid media queries**: the users are allowed to directly edit the values of the liquid media features and types, however editing these values requires some knowledge of the media queries that the majority of the users do not necessarily have. The users may also add or remove constraint instead of just editing the values.

• **Ask for permission when a new redistribution is computed**: the users can prevent scheduled migrations as they need to give permission to the algorithm before applying the *migration plan*. Asking for permission would also prevent that components are migrated to devices the user does not own, hence enhancing privacy.

• **Pin components**: the users can decide that some components should never be migrated, because they are satisfied with the current deployment of a component on a particular device. In this case the users should be able to pin the components to the devices, and the algorithm should exclude those components from the redistribution process, unless the device instantiating the component disconnects.

• **Switch from automatic to manual controls**: the users can prevent any further recomputation of the redistribution and switch to manual controllers. This is already feasible in Liquid.js.

• **Memorize usage patterns**: the redistribution algorithm can learn from the users their favourite deployment patterns, e.g., if the users move a component multiple times to the same target device, the application in the future can automatically deploy the component to the corresponding device when it becomes available.

7.6 Privacy and Security

As liquid Web applications run across multiple devices owned by different users, it becomes important to study their security [118] and privacy [114] implications so that users remain in control of their data and their devices [127]. As traditional Web applications rely on data stores mostly deployed in the Cloud, one option for achieving seamless migration and synchronization of liquid Web applications across multiple devices would also be to rely on centralized data stores, which would mostly run outside of the control of the users, thus trading off the privacy of user's data against the convenience of having a highly available and reliable storage service in the Cloud [134].

We propose instead to follow a decentralized approach both for the migration and for the synchronization, since the state of a running Web application is

directly migrated to another device using the WebRTC P2P protocol. This way, there is no data leaked outside the set of devices owned by the users as the Web application flows between them. When multiple users are involved, it becomes important to establish trust between them and their devices on which the liquid Web application will be deployed. To do so we take advantage once more of the existing WebRTC protocol and discuss how the signaling server can be extended to secure device discovery and pairing.

Liquid Web applications provide support for the following three use case scenarios [132]: • **Sequential screening**: a user owns multiple devices and starts working on one of them, eventually he decides to move the work on another (more comfortable) device. The application and the associated runtime state has to be seamlessly migrated to the other device following the user's attention focus. • **Parallel screening**: a user owns multiple devices on which the liquid application is deployed. The user may decide at any moment to change the number of the devices he is using to run the liquid application as well as to move components of the application from a device to another while keeping their state fully synchronized. • **Collaborative screening**: either a sequential screening or parallel screening scenario in which devices are owned by multiple users. State synchronization become more challenging as multiple users interact with the application concurrently and it becomes important to limit the movement and control the location of the application's code, its runtime state and the user's data. While in the first two scenarios the focus of the DAC system deals with the discovery and pairing of the devices and in the creation of the methods and infrastructures able to make an application liquid, in collaborative scenarios we also have to address *data privacy* and *security*.

• **Privacy**: as users produce their own data working with their applications, they want that it can be accessed, viewed, and modified only by the people they trust. Multi-user liquid Web applications need to be designed in such a way that the users can protect themselves from spreading their data to other un-trusted users;

• **Security**: devices are physically separated from one another and data is exchanged through standard Web protocols. We need to prevent attacks from malicious users and prevent propagation of malicious behaviours if applications are hacked.

We discuss the design of a capability-based security model in order to create a DAC [130] enabling data protection and privacy, in which users can manage the permissions able to prevent data flowing from their personal devices to the devices of un-trusted users. Likewise, users need to limit and control which liquid Web application components gets dynamically deployed on their devices coming

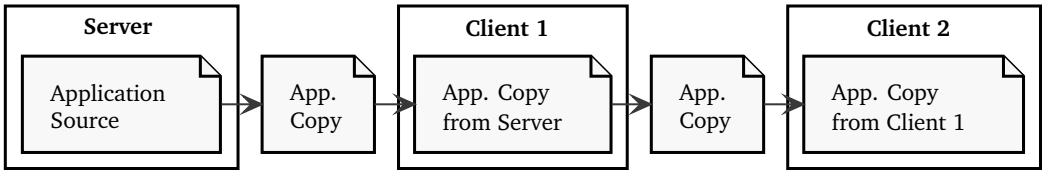


Figure 7.18. Distribution of an application in Liquid.js. The Web server(s) owns a master copy of the assets of the application. Initially, since nobody, but the Web server(s) owns a copy of the assets, client 1 download the assets, afterwards any other client connecting to the Web application can request the assets either to the Web server(s) or client 1.

from other users' devices.

Shifting to a decentralized or distributed paradigm, from a strongly centralized architecture, changes the way we are using the Web, implying that we also need to build new mechanisms for enforcing privacy and security in our decentralized Web application architectures. Trying to partially reduce the size of the workload of the central server and moving data from the server to the client, such as in Edge computing [164], has important privacy implications.

In typical distributed systems DAC models are already used for privatizing data [133]. Policies can be defined between users permitting interactions with each other. This simpler model does not require explicit representations of hierarchical user organizations, such as in Role-Based Access Control systems [138], hence users directly and dynamically decide what actions other specific users can perform on their data.

Within the IoT application domain, capability-based security models [74] have already been considered for dynamic evolving systems. In the fast changing and short living individual environment of the IoT, there is the need of creating manageable rules that quickly allow accessing data by different entities in the system. As the environment quickly evolves also the access control rules must evolve with the same speed, making the capability-based security model a suitable model for such systems.

In the liquid software domain, security and privacy for decentralized and distributed environments is currently mostly unexplored [60]. Moreover Liquid.js starts its life cycle with a **centralized** topology: the assets of an application are stored in a Web server or on multiple replicated Web servers, however it tries to be as **decentralized** as possible after the first client receives the assets of the application (in the form of JS files and liquid components defined in HTML5). Once the first client downloads the application, it is able to **distribute** it to other clients through P2P channels created with the WebRTC PeerConnection and Dat-

aChannel APIs [95] (see Figure 7.18). It is currently impossible to create a fully distributed environment in this kind of architecture, because clients connecting from their Web browser do not yet own a public IP address allowing others to connect back to them. For this reason a *Signaling Server* used both for discovery and for relaying messages between the peers containing the information on how they can create a direct WebRTC connection [181].

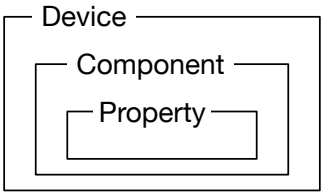
The clients in Liquid.js are thick and they are entrusted with a big portion of the logic of the Web application in order to shift the workload generated by the clients to be processed by the clients themselves instead of the central Web server. Clients can also be used as storage for both the assets and data of the application. While the topology described in Figure 7.18 and the ability of storing data directly on the clients does not generate any problem whenever all clients are owned by a single user, they do present several points of attack for malicious users whenever we are in a collaborative environment with liquid Web applications running on multiple devices shared between different users. For example whenever *client1* sends the assets of the application to *client2*, we must guarantee the integrity of the files by ensuring that the user of *client1* does not change the content of the assets themselves. Moreover users need to decide which users are trusted (e.g., *whitelisted*) and which are not trusted (e.g., *blacklisted*) to access their data by, for example, synchronizing with the current runtime state of their liquid components.

7.6.1 Privacy

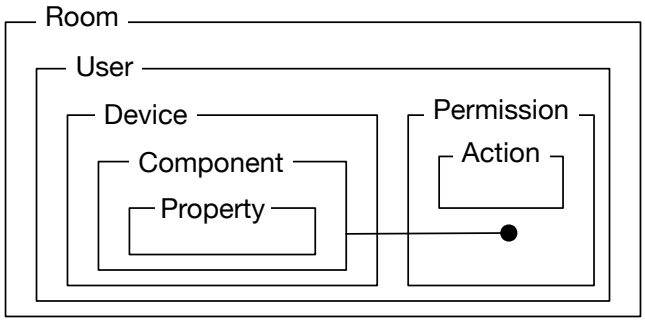
Privacy for liquid Web applications running on the devices owned by a single user can be achieved following the proximity and locality principles [114] by ensuring that the data simply does not leave the user's devices. When devices owned by different users or when devices shared by multiple users come into play, then a privacy-oriented access control system is required to ensure only authorized users can run liquid components on their devices and thus access the corresponding data. In this section we present our design for a DAC system and show how it was integrated with Liquid.js.

7.6.1.1 Design

The DAC model is based on the following entities: • **User (Individual)**: the person connected to the application; • **Data (Resource)**: sensible data owned by users; • **Action**: executable method in an application; • **Permission**: policy describing which actions a user has permit to execute.



(a) Hierarchical diagram of the entities in Liquid.js. Liquid properties represent the runtime state of a liquid component and components can only be instantiated to run on a device.



(b) Liquid.js extended with DAC model. Users own multiple devices, devices can contain a set of instantiated components, and each component contains a set of properties. Permissions can be granted to the users in order to allow them to interact with components and their properties using certain actions. Devices can join and leave rooms created by users allowing a subset of their devices to run certain components.

Figure 7.19. Extending the entities hierarchy of Liquid.js for designing Discretionary Access Control (DAC).

Figure 7.19 shows how we extend the Liquid.js framework with additional concepts for collaborative scenarios featuring discretionary access control. The *data* entity in DAC is associated with the *Component* entity in Liquid.js. The Liquid component encapsulates portions of data inside the Liquid properties. Even though Liquid properties are the smallest fragment of data in the application they are always bundled with the component encapsulating them, and the encapsulation cannot be broken. The component defines also the *purpose* of the Liquid properties, connecting them and making them inseparable whenever they are stored. This connection between sibling Liquid property makes the Liquid component itself the best target for representing *Data* in this design.

The *Action* entity in the DAC model is described inside the *Permission* entity. Liquid.js defines an API at the component and property level, granting the user the ability to interact with Liquid components. In the current version of Liquid.js

users are allowed to create, migrate, fork, clone, and pair any component in the system even though they were created by different users and reside on different devices:

• **Create**: creates an instance of a component on a device. Since the data contained in a component does not exist yet before creation, the creation of a component does not have any privacy implications. However the system protects the creation of unwanted components by imposing that only the owners of a device can create a fresh component on it. When the command is not issued by a owner of the device, then the owners must accept it manually.

• **Migrate**: moves a component (including its runtime state) to another device, at the end of the operation the original device does not keep a copy of the component. Migration requires that the owners know the identity of the receivers, and migrated components should be accepted only from known and trusted users.

• **Fork**: creates a perfect independent copy of a component on another device. Similarly to the *migration*, also fork requires that the owner trust the clients interacting with it, because they will own a copy of their data.

• **Clone**: creates a copy of a component on another device, but also keeps the state of the original and newly created component synchronized between the two devices. A cloned component has stronger privacy implications, since all its future states will be revealed to the synchronized parties. It should be possible to support both bi-directional synchronization as well as mono-directional synchronization.

• **Pair**: ensures that the state of paired components or paired properties is kept synchronized across multiple devices. This action can be performed on previously deployed components of different types and has the same security and privacy implications of *Clone*.

With the introduction of the DAC system we limit access to these actions. The *permission* entity defines a set of executable actions mapped to the liquid primitives of Liquid.js.

The entity *Room* is not directly associated with any core entity in DAC, *Rooms* group a set of *Devices* owned by multiple *Users* who decide to collaborate with each other and hence have visibility of each other. A room is used both for discovery of the presence of devices and for adding an additional layer of privacy between users, in fact inside rooms it is possible to create dynamic white and black lists (see *RoomAdmission*) in which the owner of the room can decide the devices to be admitted.

Figure 7.20 describes in detail how these entities are designed in Liquid.js:

• **User**: the user entity keeps track of all the devices the user has access to.

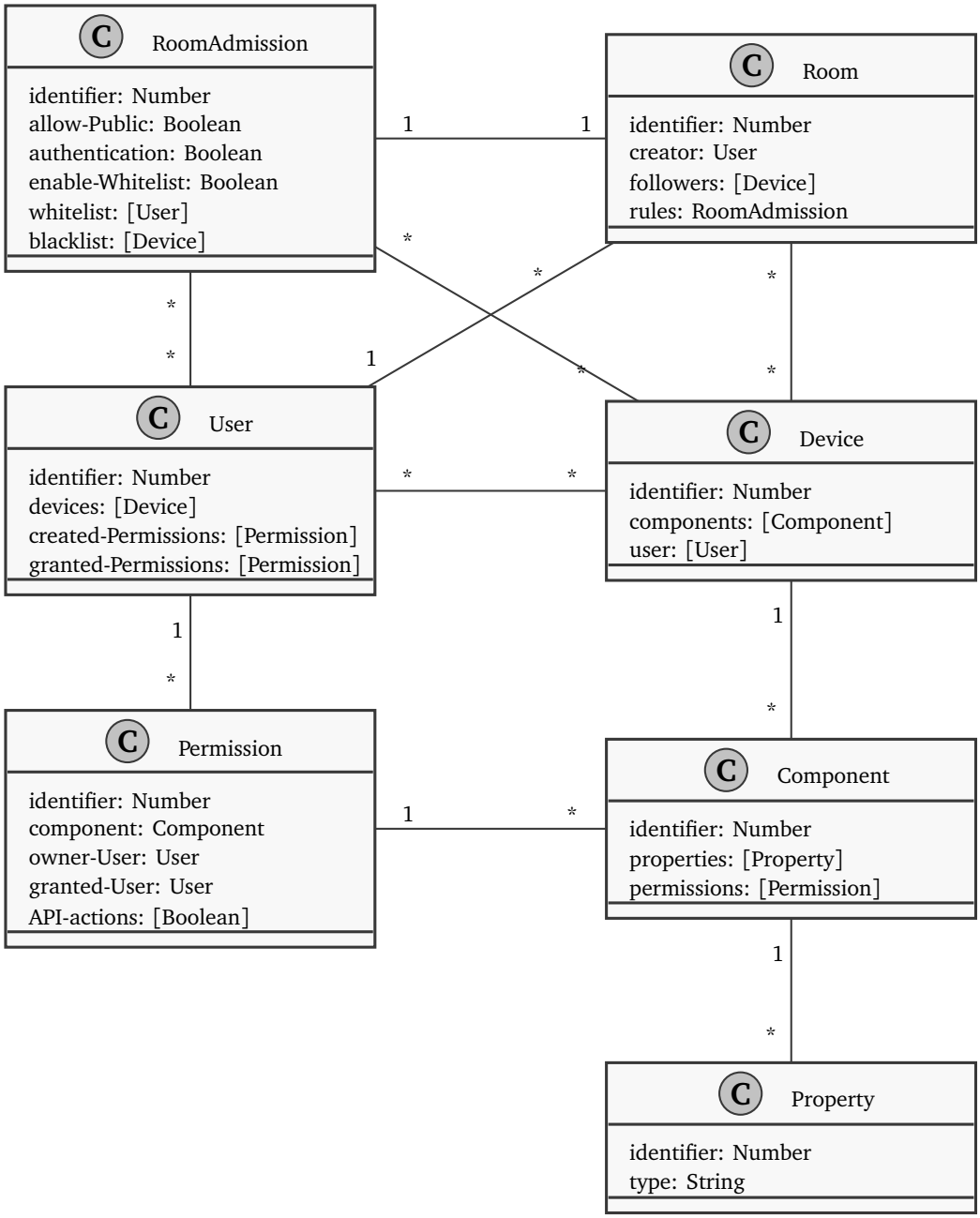


Figure 7.20. Extended entity diagram in Liquid.js

Whenever a user authenticates with his credentials from any device, the device is added to the user. The user can authenticate from multiple devices at once, which enables parallel screening. Users also keep track of the permissions they receive from other users and the ones they granted to other users;

```
Input  : device, rules
Output: true if device can join the room given the definition of rules
        attached to the room
if device.identifier is in rules.blacklist then
    return false
end
if rules.allowPublic == false ∧ device.users.length == 0 then
    return false
end
if rules.authentication == true then
    if rules.enableWhitelist == true then
        forall user of device.users do
            if user.identifier is not in rules.whitelist then
                return false
            end
        end
    end
end
return true
```

Algorithm 7.6.1: Accepting a new device in a room with assigned RoomAdmission rules

• **Device**: in the previous version of Liquid.js it could not be possible to annotate the ownership of the device. We distinguish three different cases in how a device can be used by users: **– private:** a device is owned by one single authenticated user at a time; **– shared:** a device is owned by multiple users who authenticate on the device at the same time; **– public:** a device is not owned by any specific user, hence it could be used by anybody, even unauthenticated users. The concept of *public* and *shared* devices is necessary in collaborative Liquid Web applications. For example users can edit their data on their own devices (e.g. edit on a tablet), however they want to display the data on a shared device (e.g., a smart television), so that they can receive feedback from the audience.

• **Room and RoomAdmission**: users can create any number of rooms, and devices can join a room by following it, the same device can follow multiple rooms at the same time. The creator of the room can change the rules attached to the room entity (Room Admission) that can be used to prevent users to join the room. The owner can decide to allow public devices to follow the room, to accept only users authenticated in the system and create white and black-lists. In white lists it is possible to state which users are allowed to access the room, in black

```
Input  : users, action, component
Output: true if action can be executed by the set of users authenticated
        in a device, given the set of permissions attached to a
        component
forall user of users do
    found ← false;
    forall permission of component.permissions do
        if permission.grantedUser == user.identifier then
            if permission.action == action then
                found ← true
            end
        end
    end
    if found == false then
        return false
    end
end
return true
```

Algorithm 7.6.2: Allowing API methods to be performed on components

lists it is possible to prevent particular devices to join the room. Algorithm 7.6.1 shows the pseudo-code for accepting devices which attempt to join a room.

• **Permission**: permissions are created and changed dynamically by users at runtime. Permissions are attached to the component they are referring to, associated with the user who created it, and sent to the user to which it was granted. The system is restrictive by default and the permissions defines the permit of executing actions to be performed on the data. Only the set of users authorized in the device in which the component was created on can add new permissions to the component itself, preventing un-trusted users to get the possibility of accessing someone else components. Algorithm 7.6.2 shows the pseudo-code for deciding if an executable action can be performed from a device or not. In the case that we want to allow public devices ($device.users \leftarrow empty$) to attempt to execute actions, we should also check that the condition $permission.grantedUser == empty$ is defined. In fact if the $grantedUser$ inside a permission is not defined, it means that the permission is granted to all users (authenticated or not).

• **Components**: components are built in a hierarchical structure, meaning that components may recursively contain other components. As permissions can be assigned to components independently of their nesting level, conflicts between

```
Input  : users, action, component
Output: true if action issued by users can be executed on a container
        component
if isContainer(component) then
    children ← getChildren(component);
    forall child of children do
        if checkPermission(users, action, children) == false then
            return false
        end
    end
    return true
else
    return checkPermission(users, action, component)
end
```

Algorithm 7.6.3: Conflict resolution for *container* Components

permissions may happen between different levels of the hierarchy. We take a conservative approach to deal with such conflicts whereby a *container* component can be shared only when all its *leaf* components (a component which does not contain any other component) inside of it permit the requested action (see Algorithm 7.6.3).

7.6.1.2 Architecture and Implementation

Figure 7.22 shows the architecture of Liquid.js extended with the previously described DAC system. We divide the framework between server-side and client-side and discuss them separately.

7.6.1.2.1 Server-side The goal of the decentralized Web [16] is to shift the workload from the server-side to the client-side as much as possible, however a central server is always required, because the current technologies impose that clients discover each other through a central meeting point. However we can benefit from this constraint: • **Data persistence**: users own multiple devices and can store information directly in their devices, however if a device is offline it cannot distribute the information stored in it to the other clients. While the users can decide wherever they can store their liquid components, permissions should be stored both in the client and in a central database as a reference. The permission are create by the users and these permissions affect only what

others can do with the data they create. Permissions can be distributed through P2P channels among users, however if the user is not connected, they should be accessible anyway. A central Database can be act as a fallback mechanism.

The same fallback mechanism can be used for the assets distribution, the assets of the liquid Web application can be received both from the server and from the clients if they own a copy.

• **Discovery and Authentication**: the server is used as the mean for discovery and for authentication of the users. The server can discard and expel the devices of malicious users from the system and can decide if two devices are allowed to create a WebRTC DataChannel between them.

This architecture makes it possible to trust the users connected to the application thanks to the Web server and allows clients to distribute permissions between them. The server can also be used as a fallback mechanism or for relaying messages exchanged between devices whenever they fail to communicate with each other directly (e.g., the Web browser does not implement the WebRTC standard).

The database component shown in Figure 7.22 stores information about the *Users* registered in the Web application and the corresponding permissions. The decision of making rooms persistent is up to the developer of the liquid Web application. It is not necessary to store rooms, nor the associated rules, in the database if by design the rooms are created dynamically and automatically close whenever all devices leave them. If the application requires rooms to continue to exist and be available even after the owner leaves, then also the *Room* and the *RoomAdmission* entities should be stored in the database.

The server can also ban authenticated devices from rooms whenever they attempt to issue too many unauthorized actions between clients. Clients can close their direct channels with other devices when adding them to the blacklist of a room. However in the case that a device keeps attempting to join the room, the server can prevent these actions and decide to expel and close the socket connection with the device, preventing it to sending any other further request to the liquid Web application.

7.6.1.2.2 Client-side Clients communicate directly with each other through the *WebSocket / LPC Handler* component. The *permission rules* component takes care of deciding which actions can be executed and which one are prohibited.

Figure 7.21 shows how permissions are initialized in a room: 1. Whenever a client joins a room, all other devices in the room are notified with the joining deviceId, the joining device receives the entire list of devices inside the room;

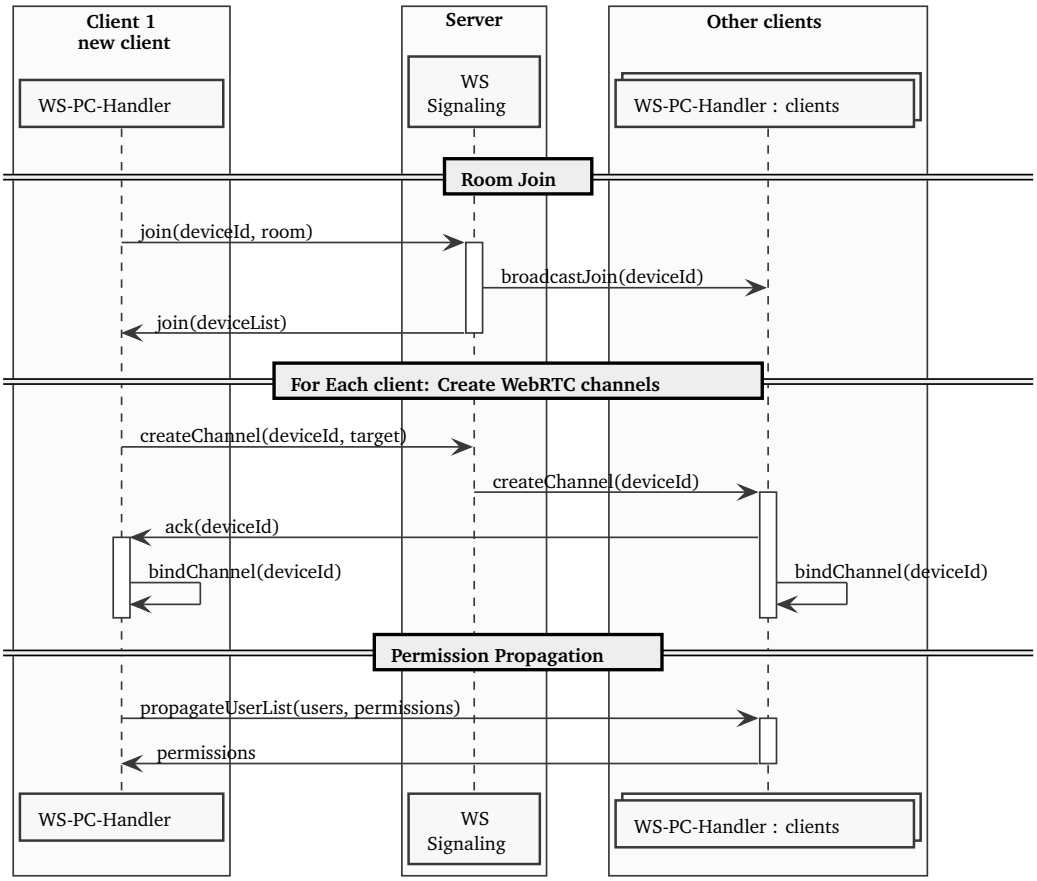


Figure 7.21. Messages exchanged during room join

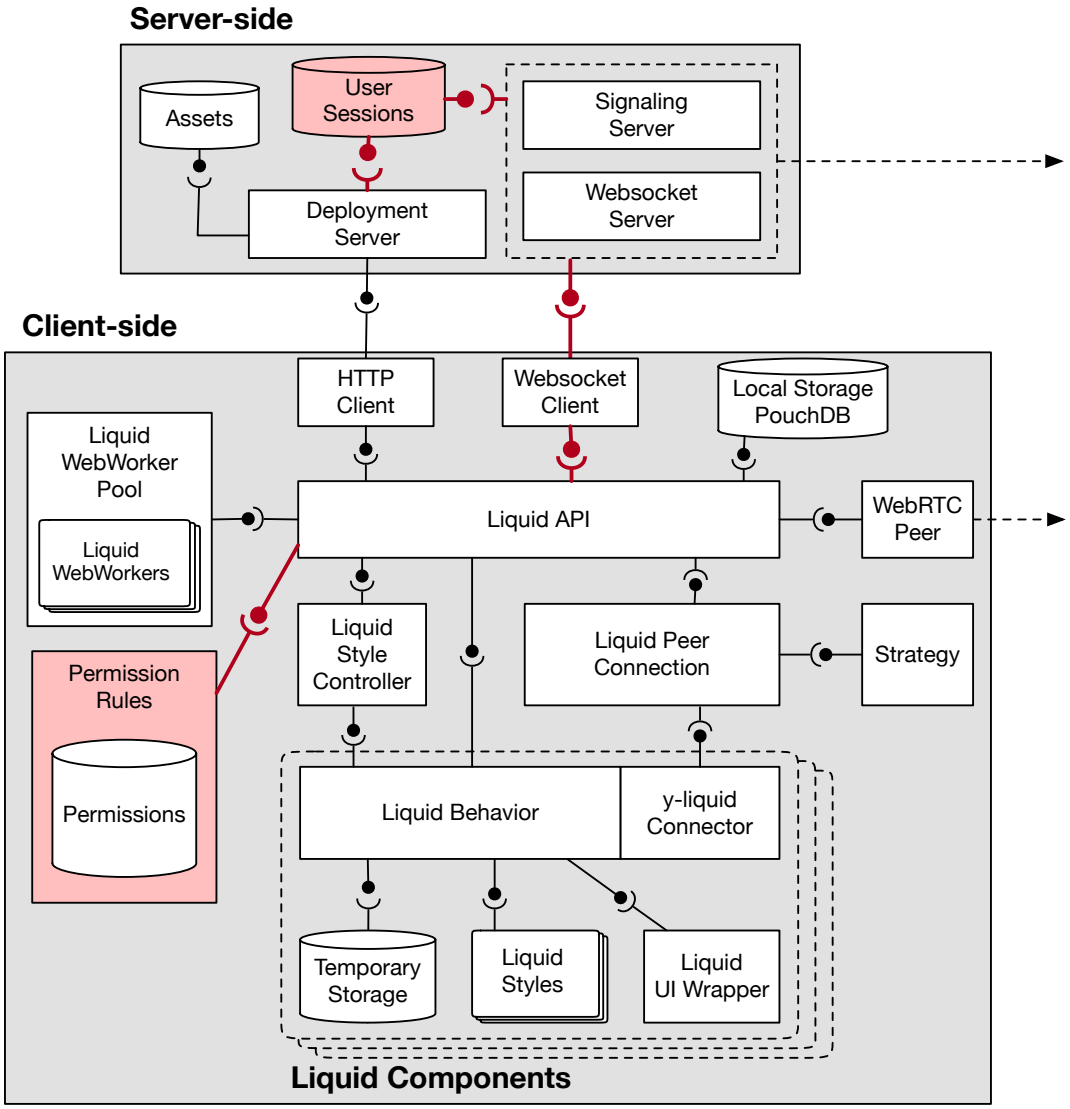


Figure 7.22. Liquid.js simplified architecture extended with the DAC model. Highlighted components and interfaces are added or changed during the extension.

2. All clients start exchanging the device descriptors of their own devices among them. The descriptor is used for initialising the WebRTC PeerConnection. In this phase clients also exchange the information about the current authenticated users in their devices; 3. The created channels are labeled with the userIds of the other clients. This binding is done locally on all devices for every channel and the label cannot be changed afterwards. This process makes sure that incoming messages passing through Peer Connections always come from a known device, meaning that devices cannot fake be to be somebody else once the channel is created; 4. Finally devices exchange with each other any permission they granted to the users in the past, making sure that they own a copy of them. The joining device does the same with the followers of the room.

Figure 7.23 shows how permissions are exchanged among devices:

1. a user creates a new permission choosing which *actions* a *user* can execute on a *component*; 2. The permission is attached to the component, only the creator of the component can attach permissions to a component; 3. The permission is forwarded both to the server-side of the application and to all devices in which the *user* is logged in; 4. The permission is associated to the owner and to the granted users inside the database. Permissions are persistent and can be requested by the users if they lose them or if they cannot access to the permissions stored in some other of their devices (e.g. because they are turned off). 5. Permissions are stored in all receiving devices. The users of the device must manually accept permissions when they are created. This process makes sure to prevent unwanted spam to happen (e.g. the migration of an unwanted component to one or more devices of an user from a malicious user's device). In this case the device can be reported to the Web server and expelled from the room.

Figure 7.24 shows how API methods can be issued on components and sent to other clients in order to be executed:

1. The *liquid API* component needs to execute an *action* on a *component* the user of device does not own. The Liquid.js API requests if the users of the device have been granted with the necessary permissions for executing the *action* on the *component*; 2. If the owner of the component granted the permission to execute the action, then the device can forward the request to the target component device, otherwise the device is encouraged not to send the request; 3. Whenever the message arrives to the target device, it checks if it is allowed to execute the received action on the component. If it does then the action is executed, otherwise the message is discarded and the peer remembers that a device tried to request an action without having permission to execute it.

It is possible to see that the process checks the permissions twice: • The first check is not required, but prevents the devices to send useless messages

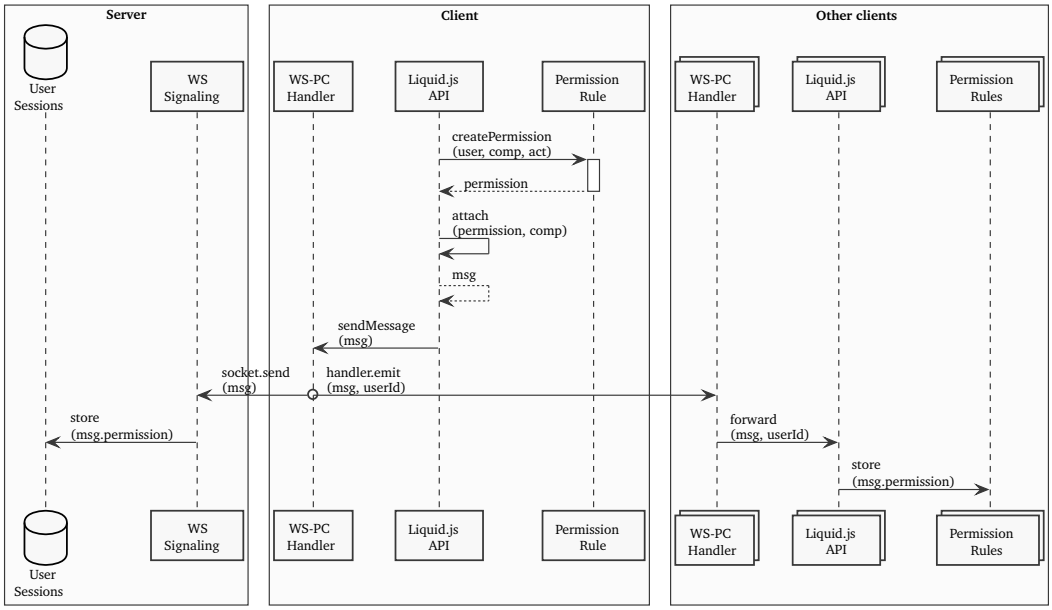


Figure 7.23. P2P distribution of the permissions

that other devices would anyhow discard. It avoids the congestion of the WebRTC DataChannel, it saves up bandwidth usage and it prevents the clients to be flagged as spammers and potentially be expelled from the room; • The second check is required, and allows users to decide which capabilities other users have access to. It prevents malicious users to access and steal their data and allows trusted users to execute only the actions they were granted permission to execute.

7.6.2 Security

Security in Web applications depends on the platform they are built upon, specifically on the Web browsers which limits access to the hardware of a device. The Web provides secure protocols which can secure the communication channels between all parties. In particular, we rely on these three protocols:

• **Hypertext Transfer Protocol Secure (HTTPS)** [154] encrypts the requests and responses sent between client and server;

• **WebSocket Secure (WSS)** [53]: WSS ensures that the persistent communication channel between server and clients is encrypted;

• **Session Description Protocol (SDP)** [140]: the SDP using the P2P transmission protocol Secure Real-Time Protocol (RSTP) ensures security over the direct WebRTC DataChannels between client devices.

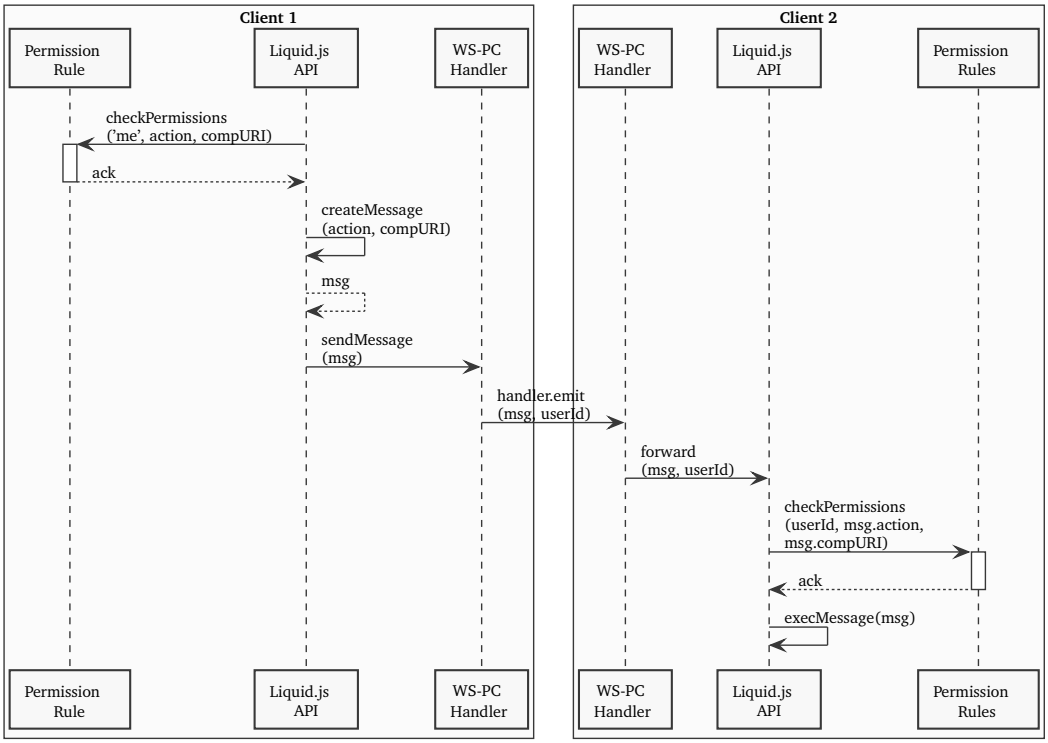


Figure 7.24. P2P issuing and checking actions

In the following sections we discuss how our design can protect users from attacks started by malicious users or protect data from attempted attacks to the system. We base our explanations starting from the security baseline provided by the protocols cited above. Malicious users may attack the platform in four different ways:

1. pretend to be the Web server of the application;
2. impersonate other users or devices identities within the peer-to-peer mesh;
3. change assets of the application at runtime and send them to other clients;
4. create malicious components attempting to run unauthorized actions.

7.6.2.1 Trusting the server

Thanks to the certificates used in HTTPS it is possible to recognize trusted server from un-trusted ones. However we do not use only HTTPS in order to reach the server, but also the WSS protocol. Figure 7.25 shows how it is possible to make sure that the same user authenticated through HTTPS requests is also the same user authenticating through WSS:

1. A client requests and receives the page from the Web server;

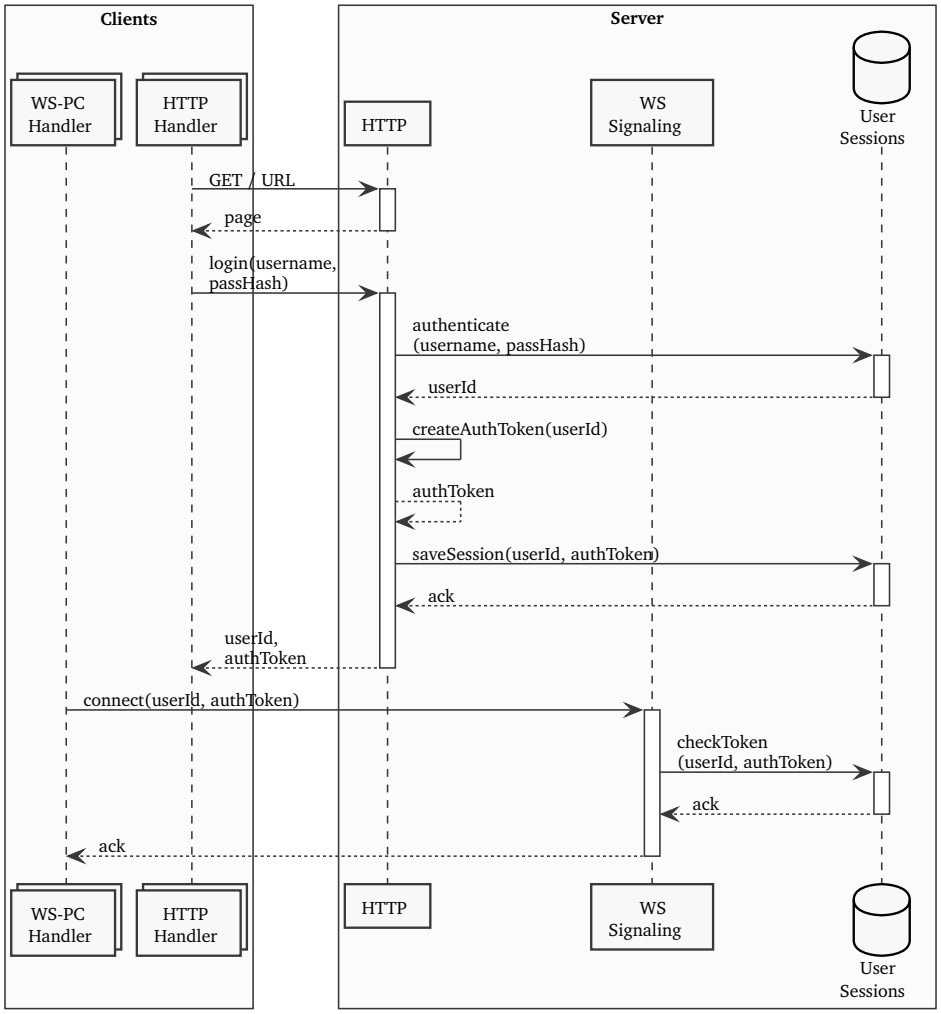


Figure 7.25. Client handshake with server

2. Afterwards the clients logs in the Web application through HTTPS. The Web server checks if the user submitted the right credentials;

3. If the users authenticate the server creates a unique *authorisation token* and stores it in the database associated with the user;

4. The client receives his own *userId* and *authentication token* back. This tuple must be sent to the Web server every single time the user must interact with the Web server.

5. The client can finally try to connect with a WebSocket connection. The request must contain also the userId and authToken.

6. The server checks if the authToken exists and confirms that the connection can be established.

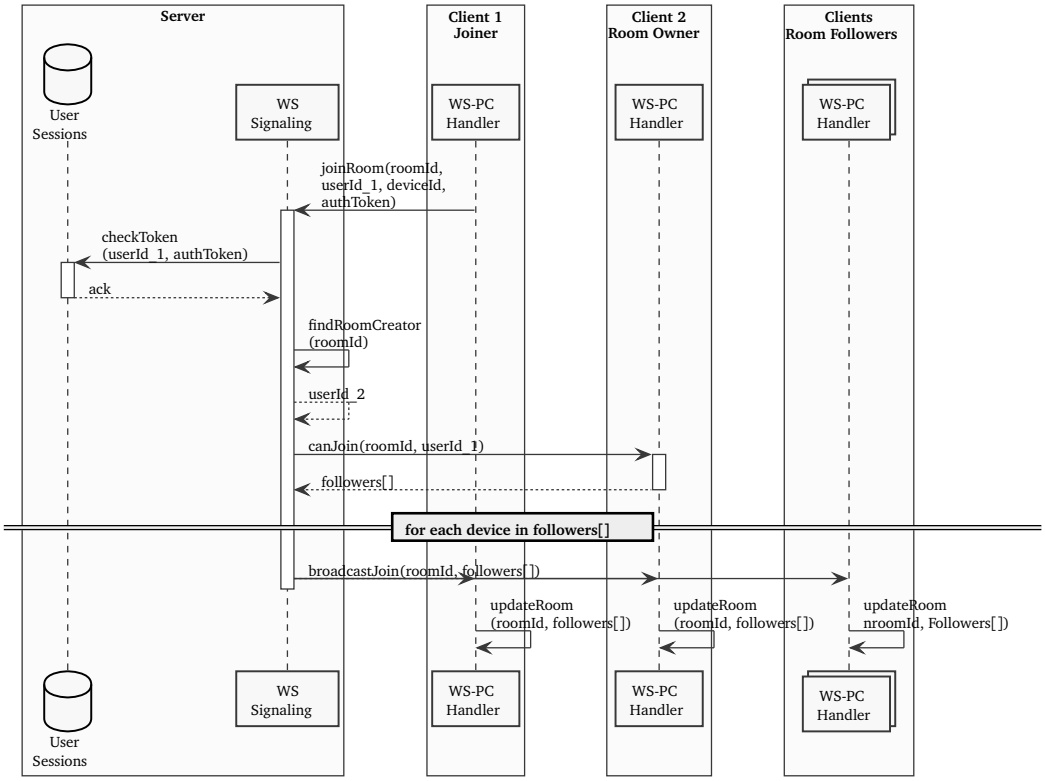


Figure 7.26. Joining a room

Since all communication with the Web server is encrypted, the authToken cannot be stolen. This ensures that users cannot impersonate someone else. HTTPS certificates ensure that all these steps are carried out with the correct server.

7.6.2.2 Trusting the client

Clients should also not impersonate other clients when they are part of the P2P network corresponding to a room. Figure 7.26 shows the process for joining a room:

1. A device can join a room if its owners know the roomId of the room they want to access, this identifier works as a shared secret between the creator and the followers of the room. The device requests to join a room using the roomId, the request contains the authToken it received when the device authenticated as well as its userId and deviceId associated with the WebSocket channel created during the handshake (see Figure 7.25);

2. The server checks the *authToken* and maps the roomId, to the creator of the room. The mapping operation can be done querying the database if the room

is persistent;

3. The server requests to the creator of the room if the user and the device are allowed to join the room;

4. If the device is allowed to join the room, the creator sends to the server the list of devices connected to the room and then broadcast to all of them the information which will allow the newly client to join the room. Thereafter the clients can exchange permissions and talk to each other without any further interaction with the server.

This process shows that the Web server initiates the creation of the WebRTC DataChannels. Since the Web server is trustworthy and the clients cannot cheat the Web server to be other users, then all PeerConnections created can be bound with the trustworthy user information (see also Figure 7.21). Once the channel is created the client cannot change their identity, meaning that the clients can trust who they are talking to.

7.6.2.3 Trusting the component

The creator of a liquid Web application may attempt to create a malicious component which automatically call API methods when loaded. For example a developer may create a component which whenever is created requests all following clients in a room to *clone* the component in their machine. While API methods can be called by the component itself and hidden from the unaware user, the *clone* action request will reach all devices, but will not be executed because the owner of the receiving device does not possess the permission associated to it. Similarly if a malicious components attempts to create permissions and then clone itself, it would not be able to overcome the manual permission acceptance given to the users whenever a permission is granted. In fact API actions called by a component can never override the manual checking done by the user. Continuous unauthorized requests will eventually result in the expulsion of the malicious device from the room. This ensures that malicious component and infected devices do not propagate and spread malicious behavior in all devices on which the liquid Web application is running.

7.6.2.4 Trusting the asset

Clients that receive assets of the application from the Web server can change those assets at runtime and attempt to infect the rooms with malicious scripts. Figure 7.27 shows how this can be prevented in our design. Every time an asset is sent through the P2P channels the clients create a MD5 checksum of the file

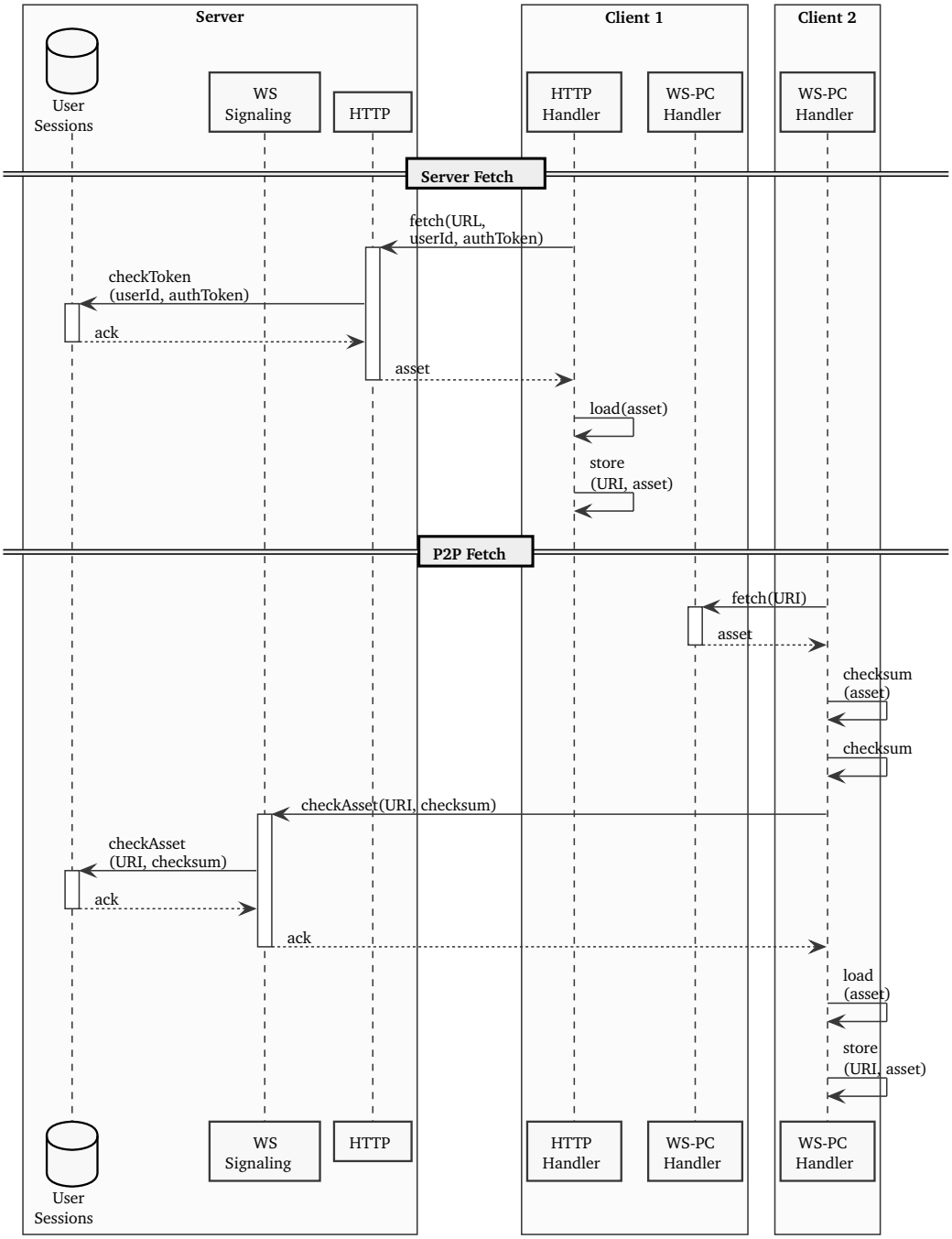


Figure 7.27. Asset fetching

and asks the server if the MD5 hash they possess is correct. If it does then it can load the assets safely, otherwise it can download the assets directly from the Web server or from another peer.

7.6.3 Limitations

While the given design can be implemented on top of native applications in any other similar component-based liquid framework, we are running our implementation of top of Web browsers, coding our scripts and components with JS. The sand-boxed environment provided by the Web browser and the interpreted nature of the JS scripting language allow users to access and change any data they posses in their Web page. Even if the data inside of the component was encrypted by the means of private and public keys, as long as it is showed in the Web page, it means it is attached to the DOM of the page and thus accessible by the user. For Liquid applications running on different environments, other than the Web browser, it is encouraged to encrypt the data inside of the component as well as the permissions stored inside of it, preventing the users to change or corrupt the permission attached on the component. Whenever a component is moved to another device, it exists as a copy of the original, if the permissions can be changed then a users may create his own copy of it and share it with anyone else. Similar malicious behavior also happen in the current design of social networks, in which it is impossible to prevent users to create a copy of someone else's data and send it to other users (e.g. uploading on Instagram a picture saved on the computer from another Instagram account).

Chapter 8

The Liquid.js API

Developers of liquid Web application need to control how to expose the liquid behavior of their cross-device Web applications to the users. The Liquid.js API gives to the developers fine-grained control over the LUE primitives, device discovery, and the life-cycle of liquid Web components.

The core Liquid.js API deals with device configuration and discovery, controls the liquid component life-cycle and exposes the LUE primitives that can be used for advanced customization scenarios of the default user experience controls provided by Liquid.js. Additionally, the API offers a cross-device version of many useful HTML5 APIs, such as LWWs, for offloading computationally intensive tasks across devices, Liquid Storage, for managing the runtime state of components shared across multiple devices, and Local Persistence, for storing snapshots of component state. The Assets API supports peer to peer deployment of the Web application assets and the Connection API provides a decentralized event bus. Many of the described methods are asynchronous because they require inter-device communication. In this case, they return Promises to represent the successful or failed completion of the asynchronous method invocation.

8.1 Framework Configuration API

The Framework Configuration API (see Table 8.1) allows developers to configure the client-side of Liquid.js and instantiate the Liquid.js framework properly. The *configure* method expects an *options* object (see Listing 8.1 for default values) in which the developer should at least define the host address of the Web server used for discovery and asset deployment.

The *getLoadableComponents* returns the list of components stored and accessible from the server-side. To enumerate the component types cached and avail-

Table 8.1. Liquid.js API: framework configuration methods

Method name and parameters	Return value
Configuration Methods	
configure(options)	Promise()
getLoadableComponents()	Promise(componentTypes[])
getAllComponentURIs([DeviceURI])	Promise(componentURIs[])
getAllComponentInstances()	componentInstances[]
getComponentInstance(componentURI)	componentInstance
getDevicename()	devicename
setDevicename(devicename)	devicename
Events - callback([values])	
devicenameChange(devicename) Triggers when the server accepts the notification of the devicename change.	
loadableComponentsListChange(componentTypes[]) Triggers when the available list of components on the server side changes and returns a list of *componentTypes[]*. The list includes only the components available on the server, and not the ones that can be obtained directly from other clients.	

able from other devices, use the Assets API.

Since it is difficult for the user to recognize a device by its *deviceURI*, Liquid.js allows developers to assign *devicenames* to the devices with the method *setDevicename*. The function can be called only on the device issuing the API method, it is not possible to change the device-name of remote devices. If the developer chooses to label devices with a name, it can replace all occurrences of *deviceURI* with the assigned device-name in all methods calls of the API. The server guarantees the uniqueness of the devicename.

The remaining methods of this API return a snapshot of the current deployment configuration of the liquid Web application. The *getAllComponentURIs* method returns either the componentURIs identifiers of all instantiated components inside the target device(s), or by default all URIs of the instantiated component on the issuing device. To access the actual components (the JS object referencing the custom element) use the *getAllComponentInstances* and *getComponentInstance*. These can only retrieve components instantiated on the device executing the command, since it is impossible to return a reference to a remote object. If the developer calls the *getComponentInstance* with an invalid componentURI, or a URIs pointing to a remote component, the value *undefined* is returned.

Listing 8.1. Configuration default options

```
defaultOptions = {
 host: 'localhost', // Web server address
 port: 80, // Web server port
 signalingServerRoute: '/signaling', // Route for accessing
     signaling server
 relayMessages: false, // Automatically relay ALL messages via the
     Web server
 workerPool: {} // Preloaded Liquid WebWorkers
}
```

8.2 Component Life-cycle API

The Component Lifecycle API (Table 8.2) are the core methods of the Liquid.js framework. Together with the LUE primitives *migrate*, *fork*, *clone* (see next Section 8.3) can be used to implement customized LUEs. The LUE primitives themselves are a pipelined composition of the methods described in this section. Exposing them in the API provides access to fine-granular mechanisms so that developers can combine them in different ways to fine-time their own LUE.

The *loadComponentType* is the first necessary step in the component life-cycle. It first checks that the assets of a component are loaded on the issuing device. If they are not yet loaded, it will request them from the Web server and dynamically load them into the Web browser. The second step on the life-cycle consists of the *createComponent* method, which creates and appends the HTML custom element tag corresponding to the Polymer component to the target DOMElement inside the DOM. The *registerComponent* takes an existing Polymer component and marks it as liquid component. If a component is not registered with Liquid.js, then any method called on this component will fail apart from *registerComponent* and *deleteComponent*. For convenience, the *instantiateComponent* method simplifies the process of instantiating a component in a single call, which is functionally equivalent to pipelining the three methods *loadComponentType* → *createComponent* → *registerComponent*. The *deleteComponent* removes the target component from the DOM and deletes it; a deleted component is lost forever as its state cannot be retrieved. The only way to save and later restore a component is to store a snapshot of its state by using the Local Persistence API (see Section 8.6).

The liquid storage for stateful component synchronization methods can be used if the target liquid component defines at least one liquid property. The *getState* method returns a snapshot of the state of a liquid component in the form of $\{propertyName : value\}$. The *setState* method allows to apply a state snap-

Table 8.2. Liquid.js API: component lifecycle and liquid storage methods

Method name and parameters	Return value
Component Lifecycle	
loadComponentType(componentTypeURI)	Promise(componentTypeURI)
createComponent(componentType [, DeviceURI, DOMElement, UIType])	Promise(componentURI)
registerComponent(componentURI)	Promise(componentURI)
instantiateComponent(componentType [, DOMElement, UIType])	Promise(component)
deleteComponent(componentURI)	Promise()
Liquid Storage for Stateful Component Synchronization	
getState(componentURI)	Promise(stateSnapshot)
setState(componentURI, stateSnapshot)	Promise(componentURI)
pairComponent(sourceCompURI, targetCompURI)	Promise()
unpairComponent(sourceCompURI, targetCompURI)	Promise()
pairProperty(sourcePropURI, targetPropURI)	Promise()
unpairProperty(sourcePropURI, targetPropURI)	Promise()

Table 8.3. Liquid.js API: Liquid User Experience (LUE)

Method name and parameters	Return value
Liquid User Experience	
migrateComponent(compURI, deviceURI [, opts])	Promise(componentURI)
forkComponent(compURI, deviceURI [, opts])	Promise(componentURI)
cloneComponent(compURI, deviceURI [, opts])	Promise(componentURI)

shot to the target component. The *pairComponent* and *pairProperty* establish a binding between two properties or between all properties sharing the same name of two different component instances so that their values will be kept synchronized thereafter. The pairing is reverted by calling either the *unpairComponent* or *unpairProperty* methods.

8.3 Liquid User Experience (LUE) API

The Liquid User Experience (LUE) API (see Table 8.3) builds upon the component life-cycle and liquid storage APIs to deliver the following three primitives [61]:

• **Migrate**: a liquid component (and its runtime execution state) is transferred from one device to another. Whenever a user performs a migrate command

on a component, he perceives that it visually moves from the source device to the target device while the original instance of the component disappears on the source device. Once the migration completes, the user can continue working on the target device resuming from the state immediately before the migration was triggered. Every time a component is migrated, the framework transparently transfers 1) the migrated component assets and 2) a snapshot of its state; the target device loads the asset if it was not already loaded, then it instantiates a new component on the target device and finally it applies the snapshot of the state sent from the source device.

- **Fork**: the fork method allows to instantiate a copy of any liquid component on a new device. From the user perspective, the source component running on the initial device is unaffected by the primitive. However, on the target device a new instance of the same liquid component appears carrying over the same state. Along with the state it had on the source device, the component carries over also the same view it was previously presenting. The copies are not connected after the command finishes executing, and the states of the original component and the forked one can evolve separately.
- **Clone**: similarly to the fork method, cloning allows to instantiate a copy of a liquid component on any target device. Differently from the fork method, the state of the original and of the cloned components is kept synchronized.

The LUE primitives are actually implemented as compositions of the *component life-cycle* methods (see Section 8.2): e.g., the *migrateComponent* method is implemented by pipelining the following methods: *connectDevice* → *getComponentState* → *getLoadedAssets* → *requestAsset* → *loadAsset* → *loadComponentType* → *registerComponent* → *createComponent* → *setComponentState* → *deleteComponent*. The pipelines defining the *forkComponent* and the *cloneCompoment* methods are very similar to the migrateComponent one, without the final call to *deleteComponent* in the case of the fork primitive, and the additional call to *pairComponent* for the clone primitive.

8.4 Device Discovery API

The Device Discovery API (see Table 8.4) allows developers to access and read the metadata related to the set of remotely connected devices constituting the execution environment of the liquid Web application. The framework fingerprints all connected devices using the ClientJS library [Jac16]. The deviceInfo object has the following form: *{deviceURI, clientjsFingerPrint, devicename, hardwareData}*. In the fingerprint we include the information about the current platform type,

Table 8.4. Liquid.js API: device discovery methods

Method name and parameters	Return value
Device Discovery	
getDeviceInfo()	deviceInfo
getDeviceURI()	deviceURI
getDevicesList()	availableDeviceURIs[]
getDevicesInfoList()	availableDeviceInfos[]
Events - callback([values])	
devicesListChange(availableDeviceInfos[]) Triggers when a new device connects to the server and is available to be paired. The event callback receives the entire list of {*deviceURI, clientjsFingerPrint, devicename, customData*} identifying each connected device.	

recognizing the following three categories: Desktop/Laptop, Tablet, and Phone. There are other possible platform values, but currently Liquid.js supports only these three, as they can run Web browsers supporting its dependencies (e.g., WebRTC, Polymer). The *getDevicesList* and the *getDeviceInfoList* ask Liquid.js to retrieve the latest version of the list of the known and currently available devices from the Web server. The *getDeviceURI* methods returns the URI of the device issuing the command.

8.5 Liquid WebWorker (LWW) API

The LWW API (see Table 8.5) is used for sharing the computational power of multiple devices to run computationally-heavy tasks by automatically offloading WebWorkers from weaker devices to more powerful ones.

The *createLiquidWorker* method allows the developer to create a WebWorker that can be shared across devices. If developers need to create multiple LWWs, they can call the method *createLiquidWorkerArray* and pass an array listing all the LWWs to be created. The purpose of the *pairDeviceWorkers* method is to establish a trust relationship between devices so that all Liquid Workers identified by the same name in the source device and in the target device can be executed replacing the other. When the *postLiquidWorkerMessage* method is called, Liquid.js will attempt to reduce the worker execution time and automatically decide whether the message should be sent to the local worker or to a remote one running on the pool of paired devices. Finally the *terminateLiquidWorker* methods ends the

Table 8.5. Liquid.js API: worker offloading methods

Method name and parameters	Return value
Liquid Worker Pool	
createLiquidWorker(workerName, workerURI)	Promise(worker)
createLiquidWorkerArray({workerName, workerURI}[])	Promise(workers[])
pairDeviceWorkers(DeviceURI)	Promise(DeviceURI)
postLiquidWorkerMessage(workerName, msg)	Promise(callResponse)
terminateLiquidWorker(workerName)	Promise(workerName)
Liquid Worker	
postMessage(msg)	Promise(callResponse)
_postMessage(msg)	Promise(callResponse)
terminate()	Promise()

life-cycle of a liquid worker.

The developer can access the Liquid worker API also without passing through the Liquid.js object, since the Liquid worker object itself exposes an API. If that is the case then the methods *postMessage* and *terminate* have the same functionalities of *postLiquidWorkerMessage* and *terminateLiquidWorker*. The *_postMessage* method bypasses the offloading functionality and ensures the task is executed on the local device.

8.6 Local Persistence API

The Local Persistence API (see Table 8.6) allows saving snapshots of the state of liquid components inside a PouchDB [Pou20] database running within the Web browser. The snapshot of the state can be saved at the device, component or property levels and any snapshot of the state can be loaded whenever the corresponding method is invoked. The snapshot is taken internally by the Liquid.js framework and does not need to be passed as a parameter to the save functions. The memento of the state is stored in JSON format, so that it can be exchanged across devices by using the event bus. The three abstraction levels allow the developer to *save* a snapshot of the corresponding state by giving the unique *key* that will be used by the PouchDB database to identify the snapshot. All abstraction levels define a *getAll* and *get* method for snapshots retrieval. Finally the device and component levels also define a *load* method which will restore on the current device the retrieved snapshot, the method will instantiate and reload the state of all liquid components contained in the snapshot. The property-level

Table 8.6. Liquid.js API: local persistence methods

Method name and parameters	Return value
Device level	
saveDeviceState(key)	Promise(key)
loadDeviceState(key)	Promise(key)
getAllDeviceState()	Promise(deviceStateSummaries[])
getDeviceState(key)	Promise(deviceStateSummary)
Component level	
saveComponentState(key, compURI)	Promise(key)
loadComponentState(key)	Promise(key)
getAllComponentState()	Promise(compStateSummaries[])
getComponentState(key)	Promise(compStateSummary)
Property level	
savePropertyState(key, propURI)	Promise(key)
getAllPropertyState()	Promise(propertyStateValues[])
getPropertyState(key)	Promise(propertyStateValue)

API does not define any *load* method because properties cannot be instantiated independently from the liquid component they belong to.

8.7 Assets API

The Asset API (see Table 8.7) is used to request and load asset files. In order to create a distributed environment that relay on the Web server as little as possible, Liquid.js allows clients to exchange asset files among one another. To make this possible, at least one connected client needs to own a cached copy of the assets initially stored on the Web server. For security reasons not all assets can be shared using the Asset API, the list of shareable assets must be filled in a configuration file. Assets can be shared only on-demand, clients cannot send assets directly to other clients if the receiving client did not send a request. The *requestAsset* method allows the developer to poll a device for a specific asset which can then be executed on the machine by calling the *loadAsset* function. The *getAsset* method retrieves the script of any asset that was previously executed on the machine, and the *getLoadedAssets* methods returns an array containing all names of executed scripts. The *loadingChange* event is associated to this API: whenever Liquid.js is in the process of requesting or loading a file from another client it will set its loading status to true, in all other cases the status is set to false.

Table 8.7. Liquid.js API: assets lifecycle methods (P2P)

Method name and parameters	Return value
Assets	
requestAsset(name, deviceURI)	Promise(script)
loadAsset(name, script, type)	Promise(name)
getAsset(name)	script
getLoadedAssets()	names[]
Events - callback([values])	
isLoadingStatusChange(status) Triggers when Liquid.js starts or ends fetching assets from the server or another client. The event carries one of the following loading status codes: *true* if Liquid.js is fetching or loading at least one asset; *false* if Liquid.js is not currently fetching or loading any assets.	

8.8 Connection API and Event Bus

The Connection API (see Table 8.8) defines all methods that can be used by the developers to communicate with other devices, or with the Web server if they need to exchange data with it. The API exposes three methods that can be used to enhance the server-client communication passing through a WebSocket channel: the *isSocketConnected* method returns the current status of the connection, the *sendSocketCustomMessage* method is used for direct communication with the server through special purpose socket messages, and the *socketDisconnect* method closes the connection with the server. The remaining four methods are used to interact with the WebRTC channels connecting clients: the *connectDevice* ask Liquid.js to open a connection between the current device and the target device, similarly the *disconnectDevice* forces to close an opened connection between target clients; the *sendMessage* method allows developer to exchange messages with other clients; and the *getConnectionList* method returns an array containing all deviceURIs of all devices that share an opened connection with the issuing device.

The device connection API triggers the *connect*, *disconnect* and *reconnect* events whenever the socketConnected status changes. Moreover the developers can define their own custom sockets events if they are developing Web applications that need to communicate directly with the Web server.

Table 8.8. Liquid.js API: device connection and event bus methods

	Method name and parameters	Return value
WebSocket and WebRTC		
Server	isSocketConnected()	isSocketConnected
Server	sendSocketCustomMessage(msg)	-
Server	socketDisconnect()	-
Device	connectDevice(deviceURI)	Promise(deviceInfo)
Device	disconnectDevice(deviceURI)	Promise()
Device	sendMessage(message)	-
Device	getConnectionList()	connectedList
Events - callback([values])		
connect(deviceID) Triggers when the client connects to the server for the first time.		
disconnect() Triggers when the client disconnects from the server.		
reconnect() When the client reconnects to a server, this event is triggered **instead of** the *connected* event.		
Custom Events		
It is possible to use the socket connection opened by Liquid.js to communicate with the server side or allow the server to push custom data to the client. By sending a *socketCustom* message defined as {*type, payload*}, liquid.js fires an event *type* with parameter *payload* on the receiving client.		

Chapter 9

Validation

In this chapter we validate the design of the three application layers and evaluate the implementation of Liquid.js. First we discuss the validation of the layers and then we show how to build liquid Web applications with the framework.

9.1 Data Layer and Synchronization

In this section we focus on the synchronization performance of the data layer of Liquid.js. We compare the performance of the different strategies presented in Section 7.3.1 and we measure how the overall LiquidPeerConnection (LPC) design impacts on the synchronization of the data layer.

The strategies are evaluated with three different kind of topologies: • **star topology**; • **spanning tree topology**; • **linear topology**. The star topology is automatically constructed by the full-graph strategy at runtime, while the linear and spanning tree topologies can be constructed with the Minimal Connection (MinCon) strategy.

9.1.1 Evaluation of the Routing Table

We evaluate the implementation of the local RT stored in the LPC component (described in Section 7.3.2.3) by comparing the relative data transfer (bandwidth) between all pairs of devices in a sample topology. The experiment is ran both when the RT is enabled and disabled, so that we can directly compare the gathered values.

Specs	Computer 1	Computer 2
Brand	MacBook Pro (Mid 2014)	MacBook Pro (late 2012)
OS	macOS Sierra v10.12.2	macOS Sierra v10.12.2
Processor	2.2 GHz Intel Core i7	2.5 GHz Intel Core i7
# of cores	4	4
Memory	16 GB 1600 MHz DDR3	8 GB 1333 MHz DDR3
Browser	Chrome v.55.0.2883.95 (64-bit)	Chrome v.55.0.2883.95 (64-bit)

Table 9.1. Specification of the two devices used for the evaluation.

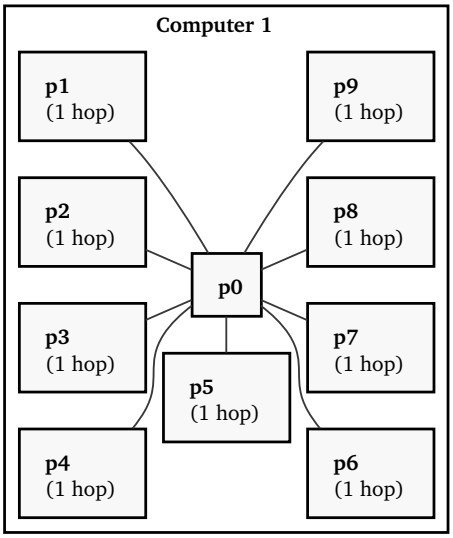
9.1.1.1 Testbed Configuration

All the data presented in this section was collected by running the test in the two following scenarios:

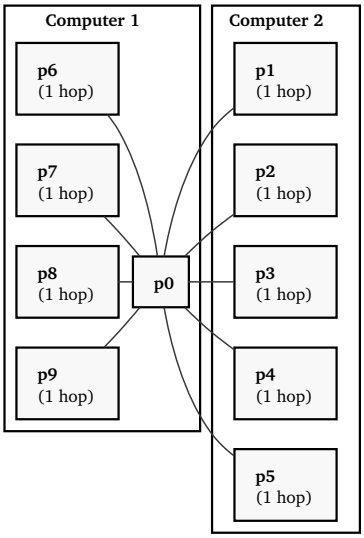
1. in all single device scenarios (e.g., when only *Computer 1* is connected), all peers are running on the same physical machine. Multiple peers can be deployed on the same machine by opening multiple Web browser tabs. Each Web browser tab connects to the application with a different `id`, meaning that they behave like different devices, even if the peers are deployed locally on the same machine. In the case the peers are deployed on the same physical host, each peer still relies on WebRTC DataChannels to deliver messages to all other peers. The specifications of *Computer 1* can be found in Table 9.1;

2. in the multi-device scenarios (e.g., when both *Computer 1* and *Computer 2* are connected) peers are distributed among two different physical hosts. Similarly to the single device scenarios, when multiple peers need to be deployed on a single machine, they are simulated with multiple tabs of the same Web browser. The WebRTC DataChannels decide when messages are sent locally or through the network. The specifications of *Computer 2* can be found in Table 9.1. The two computers are connected to the same wireless local network using a 5 GHz frequency in 802.11n mode.

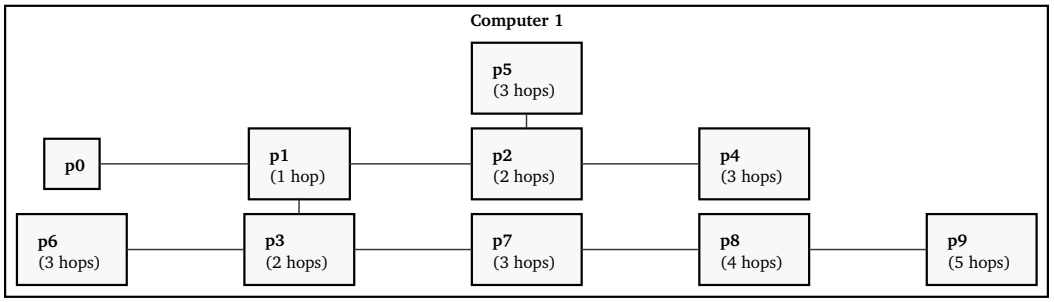
In order to provide as much consistency as possible, for each new run the devices were restarted. The Web browser tab where peer `p0` was always the one on focus until the experiment finished (the *onFocus* tabs has higher priority). All the machines started the experiment under the same conditions and their memory was free.



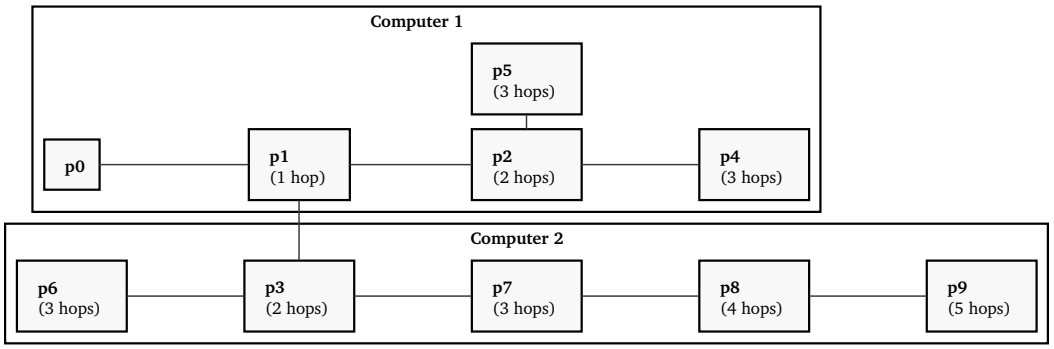
(a) Network visualization of the topology in the full graph single device example.



(b) Network visualization of the topology in the full graph on multiple devices.



(c) Network visualization of the topology in the MinCon single-device example.



(d) Network visualization of the topology in the MinCon multi-device example.

Figure 9.1. Topologies used in the evaluation of the data layer of Liquid.js. The number of hops is computed relatively to the peer `p0`.

P#	Hops	Single Computer Up [MB/s]	Single Computer Ping [ms]	Two Computers Up [MB/s]	Two Computers Ping [ms]
P0	-	-	-	-	-
P1	1	30.83	8	18.38	9
P2	1	29.11	7	16.10	10
P3	1	28.31	7	18.78	10
P4	1	33.29	7	18.09	9
P5	1	29.67	8	17.76	15
P6	1	34.9	7	37.11	7
P7	1	34.55	7	34.09	8
P8	1	30.62	8	32.36	8
P9	1	32.38	7	34.27	7

(a) Performance of the fully connected strategy.

P0	-	-	-	-	-
P1	1	29.58	8	32.27	7
P2	2	15.74	20	17.13	25
P3	2	13.99	17	12.26	27
P4	3	9.29	39	12.07	58
P5	3	9.77	39	11.75	68
P6	3	10.22	43	8.55	56
P7	3	9.46	42	7.96	71
P8	4	6.65	67	6.36	133
P9	5	5.25	93	5.15	145

(b) Performance of the MinCon strategy without routing table.

P0	-	-	-	-	-
P1	1	31.59	8	34.58	8
P2	2	15.91	11	15.79	9
P3	2	16.33	10	12.91	26
P4	3	11.24	17	11.40	14
P5	3	11.12	15	12.29	12
P6	3	11.20	15	8.21	33
P7	3	9.72	12	9.00	29
P8	4	7.43	21	6.34	33
P9	5	5.59	27	4.98	36

(c) Performance of the MinCon strategy with routing table.

Table 9.2. Experimental values measured on all peers ranging from `p1` to `p9` based on the topologies shown in Figure 9.1. Peer `p0` starts the conversations across all peers. White rows represent peers deployed on Computer 1, dark rows represent peers deployed on Computer 2.

9.1.1.2 RT in Sequential Conversations

The topologies used in this experiment are composed by 10 interconnected peers as shown in Figure 9.1. All diagrams show the numbers of *hops* relative to peer `p0`, which is also the peer that starts the conversation and initially start exchanging messages across the peers. In Figure 9.1c we show the topology created with the MinCon strategy, when all peers are deployed on a single device, while in Figure 9.1d we show the same topology deployed on two computers. In the latter scenario, five peers are deployed on each device. The full-graph strategy is used as a baseline for the message passing comparison, since we expect it to be the topology with the lowest latency and the highest relative bandwidth, as there is no need for the LPC to query the RT and all the peers can exchange messages directly. In Figure 9.1a we show the topology of the full-graph strategy deployed on a single device and in Figure 9.1b we show the topology deployed on two devices.

In Table 9.2 we show all the results of the evaluation. Each cell is computed as the average of five independent runs. The upload bandwidth and the ping are all relative to peer `p0`, even if there is not a direct connection between them. The bandwidth is computed locally in `p0`'s LiquidPeerConnection (LPC) component and represents how much data flowed between two devices since the moment the message is created in `p0`'s LPC and until it was processed in the target device's LPC[1]. Since `p0` is the peer that starts the conversation, we ignore to compute the ping and bandwidth for sending messages between components deployed on `p0` itself, because the internal message passing does not requires to query the LPC component. The size of the message payload transferred between the devices is 270 Kilobytes.

It is important to note that in this section our goal is to evaluate the impact of the RT in the message transferring, **we are not evaluating the performance of the synchronization across the devices**. In this experiment messages are passed sequentially, and not in parallel. E.g., `p0` first sends a message to `p1`, then, once the message reaches `p1`, it sends a message to `p2`, and so on. In Section 9.1.1.3 we will talk about parallel conversations, and in Section 9.1.2 we evaluate the actual parallel Yjs synchronization implemented in Liquid.js.

In Table 9.2a we show the results for the full graph strategy where all peers are 1 *hop* away from `p0`:

[1]The bandwidth we compute is not the network bandwidth and we are not computing the maximum rate of data transfer of each individual peer. We are computing the rate of data transferred between devices during the experiment from the point of view of `p0`. The data transfer lapse also includes the LPC overhead, the RT querying and the packaging process when enabled.

• In the single-device scenario the average upload bandwidth is 31.5 Megabytes/second. It is interesting to note, that even if the peers are deployed on the same physical machine, the processing time of the ping messages in both machines takes a total time of 7.3 milliseconds on average. The latency between peers deployed on the same machine is high because nevertheless the ping messages pass through the WebRTC DataChannel and are queued twice in the respective DataChannel's queues. Since the ping message is processed as soon as the DataChannel triggers a *onPing* event, the 7 milliseconds represent both the time spent in queues and the time needed for triggering the event. For each individual peer there is no significant difference in the gathered values.

• In the multi-device scenario we can clearly see that the average upload bandwidth decreases to 25.22 MB/s. More precisely, the local average upload bandwidth on *Computer 1* increases to 34.35 MB/s, while the remote average upload bandwidth on *Computer 2* lowers to 17.82 MB/s. The bandwidth on *Computer 1* increases because fewer peers are deployed on the same machine, allowing a faster processing of the messages. However, the total time-lapse of the data transfer and message processing of the 270 KB message from *Computer 1* to *Computer 2* is almost doubled, therefore decreasing the relative bandwidth between the two machines by almost half (51.71%). The average ping of the peers deployed on *Computer 2* is 10.6ms, an increment of 3.3ms from the single device deployment. These milliseconds represent the time spent in transit from a device to another in the local WiFi network.

In Table 9.2b we show the results for the MinCon strategy without using the RT. In this evaluation the distance between the peers is not constant and depends on the topology shown in Figure 9.1c and 9.1d):

• In the single-device scenario the average bandwidth decreases as the number of hops increases. The average upload bandwidth from `p0` to `p1`, which is 1 *hop* away from `p0` is 29.58 MB/s, a value consistent with the previous experiment shown in the full graph scenario. The average upload bandwidth for the peers 2 *hops* away from `p0` is 14.87 MB/s, 50.25% of to the 1 hop bandwidth. The average bandwidth for the peers 3 *hops* away is 9.685 MB/s, 32.74% of the 1 hop bandwidth, the bandwidth for the 4 *hops* is 6.65 MB/s, 22.48% of the 1 *hop* bandwidth, and the 5 *hops* is 5.25 MB/s, 17.75% of the 1 *hop* bandwidth. These values are not surprising, because each peer relays messages to the next peer in the topology, in fact we expected to see exponential decrements of the bandwidth as the number of hops increased. E.g., when a message needs to be delivered from `p0` to `p2`, `p0` initially sends the 270 KB message to `p1`, which then forwards it to `p2`. The total size of all messages sent by the peers in the topology for reaching `p2` is 540 KB, which is double the size of KB exchanged when we

	Full Graph				MinCon			
	Upload		Download		Upload		Download	
P#	# msg.	Size [KB]	# msg.	Size [KB]	# msg.	Size [KB]	# msg.	Size [KB]
P0	9	2430	0	0	9	2430	0	0
P1	0	0	1	270	8	2160	9	2430
P2	0	0	1	270	2	540	3	810
P3	0	0	1	270	4	1080	5	1350
P4	0	0	1	270	0	0	1	270
P5	0	0	1	270	0	0	1	270
P6	0	0	1	270	0	0	1	270
P7	0	0	1	270	2	540	3	810
P8	0	0	1	270	1	270	2	540
P9	0	0	1	270	0	0	1	270
Total:	9	2430	9	2430	26	7020	26	7020

Table 9.3. Number of messages and data transferred for each peers in the examples shown in Table 9.2 without the RT. The size of the messages exchanged only counts the payload of the synchronization messages without considering the headers of the wrapper. The wrapper contains the URI of the destination, the sender URI, timestamp and other metadata used by the LPC component, for a total of on average 60 bytes.

compare it to th full-graph strategy for reaching the same peer. By doubling the size and number of messages, we expected to observe a decrease of 50% in the relative bandwidth. Similarly we expected to observe 33% bandwidth for the 3 *hops* peers, 25% for the 4 *hops* and 20% for the 5 *hops*. In our experiment the percentages are slightly lower than those, because of the overhead derived from relaying the message. In fact whenever a message reaches a peer, the peer needs to decide if the message should be forwarded to another peer in the topology. When the peer reads a message from the WebRTC DataChannel queue, it decides if it has reached the destination or if it should query the LPC and the strategy component for the next target. This process is the cause of the overhead in the data transferred. In Table 9.3 we show the number of messages and size of data sent and received by all peers: in the full graph example `p0` sends a total of 2'430 KB and 9 messages, while all other peers received 270 KB; in the MinCon experiment, where the peers are required to relay messages, the total number of messages transferred is 26, 2.8 times more data than the full graph strategy. Given the same number of peers, if the topology constructed by the MinCon strategy is linear (similarly to the topology we discuss later in Figure 9.3), the total size of messages transferred is 45, 5 times more messages than the full graph. In the best case, in which the MinCon strategy constructs a binary-tree-like topology with root `p0`, the total number of messages transferred is 18, 2 times more

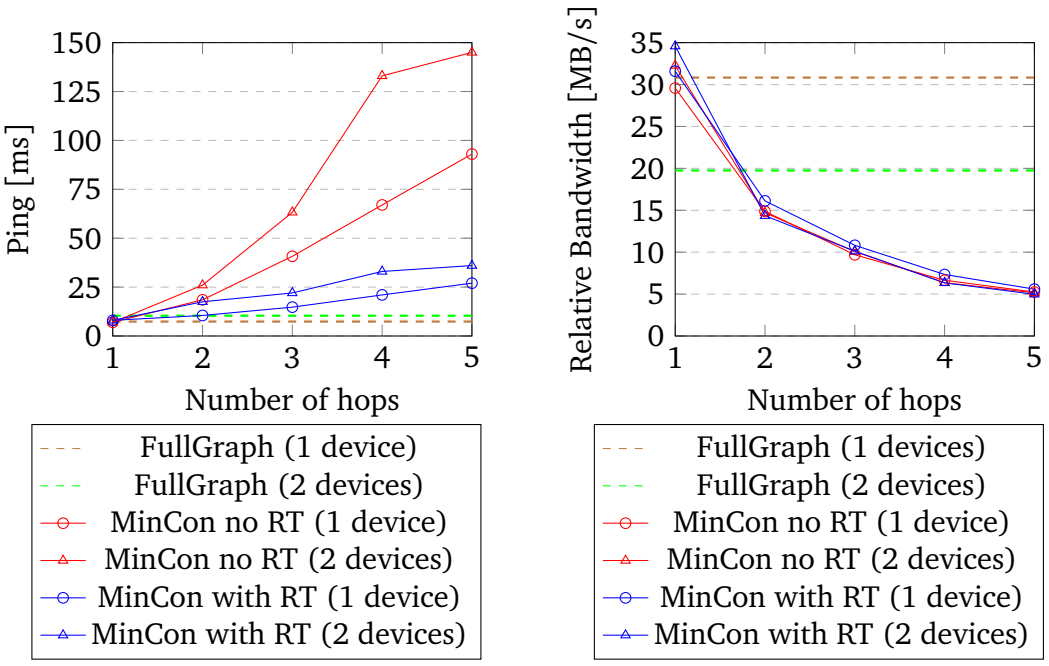
messages than the full graph.
The increase in the average ping values in the experiment are the following: 8 ms for 1 *hop*; 18.8 ms for 2 *hops*, 2.3 times the value of 1 *hop* ; 40.75 ms for 3 *hops*, 5 times the 1 *hop* ; 67 ms for 4 *hops*, 8.3 times the 1 *hop*; 93 ms for 5 *hops*, 11.6 times the 1 *hop*. These values are much worse than we could expect, since we would expect a linear increase. The exponential growth is due to the fact that we are not using the RT, and thus before forwarding a message to a peer, the network is flooded with probing messages.

• In the multi-device scenario we can infer similar trends. As expected, the ping increases when the peers are deployed on remote devices. If we compare the average ping of the single-device scenario with the pings of the peers deployed on a remote computer in the multi-device experiment, we see that the increase is constant: the average ping of the peer 2 *hops* away is 27 ms, 1.45 more than before; 63.25 for 3 *hops*, 1.55 time more; 133 for 4 *hops*, 1.98 times more; and 145 for 5 *hops*, 1.55 times more. We can also see that the average ping for local devices has increased and adapted to the value of the ping of the peers deployed on the remote device. Again this is due to the fact we are not using the RT, thus the MinCon strategy recomputes the path to a peer every time a message is transferred, affecting the ping computation which waits for all the responses sent by all other peers. We will see that the RT will significantly improve the ping performance in the next experiment. The average upload bandwidth has a constant drop when compared to the single-device scenario. In average the bandwidth with peers deployed on a remote device is 9% lower.

In Table 9.2c we show the results for the MinCon strategy with the RT. In this experiment the distance between the peers is not constant and depends on the topology shown in Figure 9.1c and 9.1d):

• In the single-device scenario we see that the RT impacts positively both on the bandwidth and ping. The average ping grows linearly and starts from 8 ms and increases on average by 4 ms for each *hop*. The RT allows to skip the querying of the strategy component, which fastens the execution and relaying of a message inside the LPC component. The average upload bandwidth also increases when compared to the evaluation of the MinCon without RT: for peers 2 *hops* away the average bandwidth is 16.12 MB/s, 8% faster than before; for 3 *hops* the average is 10.82 MB/s, 11% faster; for 4 *hops* it is 7.43 MB/s, 11% faster; and for 5 *hops* it is 5.59 MB/s, 6% faster.

• In the multi-device scenario we have similar results. The ping grows linearly and the average 5 *hops* ping is 36 ms, almost 4 times lower (24.8%) if compared to the experiment without RT. The average 5 *hops* bandwidth is 4.98 MB/s.



(a) Ping comparison

(b) Relative bandwidth comparison

Figure 9.2. Comparison between full-graph, minimal connection with and without routing table,the topologies shown in Figure 9.1. For helping with the comparison, the full-graph strategy is shown with a prolonged dashed line of the 1 hop values.

In Figure 9.2 we plot the results. We compare the average values of each experiment for both the ping and bandwidth. In Figure 9.2a we compare the average ping of the six experiments. On the x-axis we display the number of *hops* and on the y-axis the ping in milliseconds. The dashed lines represents the average values of the full-graph strategy for both single and multi-device experiments. The full-graph strategy has the best ping overall, while the MinCon strategy with RT has a better trend compared to the experiments without the RT. In Figure 9.2b we plot the average bandwidth in MB/s (y-axis) for all six experiments. All MinCon experiments have a similar trend, but the RT experiments are slightly faster. As expected, we can see that the fully connected graph outperforms the MinCon strategy both in terms of ping and bandwidth for both the single and multi-device experiments. We can also see that the enabling of the local routing table greatly reduces the ping between the machines.

	Full Graph				MinCon			
	Upload		Download		Upload		Download	
P#	# msg.	Size [KB]	# msg.	Size [KB]	# msg.	Size [KB]	# msg.	Size [KB]
P0	9	2430	0	0	1	270	0	0
P1	0	0	1	270	1	270	1	270
P2	0	0	1	270	1	270	1	270
P3	0	0	1	270	1	270	1	270
P4	0	0	1	270	1	270	1	270
P5	0	0	1	270	1	270	1	270
P6	0	0	1	270	1	270	1	270
P7	0	0	1	270	1	270	1	270
P8	0	0	1	270	1	270	1	270
P9	0	0	1	270	0	0	1	270
Total:	9	2430	9	2430	9	2430	9	2430

Table 9.4. Number of messages and data transferred for each peers in the examples shown in Figure 9.1 when the RT is enabled in a parallel execution environment. The size of the messages exchanged only counts the payload of the synchronization messages without considering the headers of the wrapper. The packaged message header contains the URI of all destinations, the sender URI, timestamp and other metadata used by the LPC component, for a total of on average 80 bytes plus 14 bytes for each destination after the first one.

9.1.1.3 Decreasing the Messages Transferred with Message Packaging in Parallel Conversations

In the previous section we showed the effect of the RT in the sequential data transfer using different strategies. The consequence of using the MinCon strategy is that messages must be relayed through other peers in order to reach the final destination. In a parallel scenario, the network would be congest with multiple messages, decreasing the performance of the synchronization process even further. If the synchronization process takes too long, then the updates of the liquid state would not be visible on real-time on companion devices, and thus diminishing the effect of the expected LUE responsiveness. The message packaging feature allows the LPC to send multiple copies of the same payload to multiple destinations using a single message.

In Table 9.4 we show the effect of the message packaging during the parallel synchronization of the topologies shown in Figure 9.1. The results of the parallel conversation with the packaging feature enabled, are similar to the sequential one without packaging. Locally, on the source peer that creates the packaged message, we computed that the packaging feature has an overhead of 1 millisecond when we compared it with the transfer without the same feature. On all

target machines, the unpacking has no overhead when compared with the sequential synchronization, because the unpacking of the message from the queue was already necessary and happening in the sequential experiment in order to read the target destination stored in the relayed message headers.

9.1.2 Evaluation of the Yjs Synchronization

In Section 9.1.1.2 and in Section 9.1.1.3 we evaluated the RT of the LPC with a simple conversation: `p0` sends the payload containing the updates of the state to another peer and the peer updates its own state locally, no further communication happened. The algorithm used by Yjs [139] for synchronizing data between peers has a more complex conversation than the one we evaluated in the previous sections. Whenever a peer updates its own state, it queries the Yjs component for creating the update payloads and then it sends them to all connected peers with the help of the LPC. This process however does not finish when the payload arrives to the peers and the update is patched. In fact, once the update is patched, Yjs queries all other connected peers and asks them if they are currently working on the last version of the state. If one peer, for any reason, is not up to date, Yjs decides if it owns a newer version of the state, or if it should be patched and repeat the previous process until all peers are working on the last version of the state. The conversation requires all peers to communicate multiple times in order to ensure that the state between them is consistent. The consequence is that the synchronization process finishes only when all peers update their state and transmit all messages to all other peers. For this reason the duration of this process depends directly on the bandwidth and latency of the slowest peer connected in the topology.

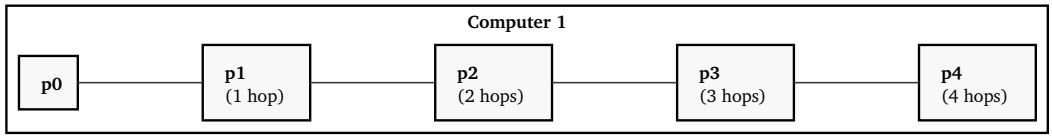
In this section, we compare the relative bandwidth of the Yjs synchronization happening in the full-graph topology against the relative bandwidth performed by the MinCon strategy. We start the evaluation in Section 9.1.2.2 by showing the impact of the payload size and message packaging feature on the synchronization process on a linear topology, then in Section 9.1.2.3 we show the impact of the RT and message packaging features on the synchronization performance on the topology previously presented in Figure 9.1. The evaluation will both address single and multi-device deployments.

9.1.2.1 Testbed Configuration

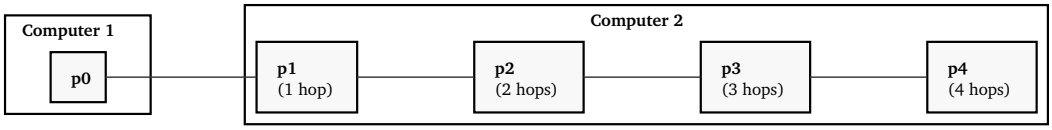
In Table 9.5 we show the specifications of the devices used in the evaluation. All the experiments follow the same rules and conditions explained in Sec-

Specs	Computer 1	Computer 2
Brand	MacBook Pro (Mid 2014)	MacBook Pro (late 2012)
OS	macOS Sierra v10.12.2	macOS Sierra v10.12.2
Processor	2.2 GHz Intel Core i7	2.5 GHz Intel Core i7
# of cores	4	4
Memory	16 GB 1600 MHz DDR3	8 GB 1333 MHz DDR3
Browser	Chrome v.55.0.2883.95 (64-bit)	Chrome v.55.0.2883.95 (64-bit)

Table 9.5. Specification of the two devices used for the Yjs synchronization evaluation.



(a) Deployment on a single device.



(b) Deployment on two devices.

Figure 9.3. Linear topologies used in the Yjs synchronization evaluation. The number of hops is computed relatively to the peer `p0`

tion 9.1.1.1.

9.1.2.2 Payload Size and Message Packaging in Linear Topologies

In Figure 9.3 we show the linear topologies we used for both the single and multi-device evaluations. We compare the two deployments against each other and against the full-graph strategy. In Table 9.6 we show the results of the evaluation. For each row, we run the experiment 5 times and in each run we update the state of a liquid property on `p0` 10 times with an interval of 30 seconds between an update and the other. We run the experiments with 270 KB and then 5 MB payloads, in order to check if the size of the message has an impact on the performance of the synchronization. The values displayed on the tables are the average of all relative upload bandwidth values computed on `p0` when the synchronization process ended for each of the 10 updates in the 5 runs. Since the synchronization process depends on the slowest peer in the topology and not only on the distance (in hops) between the source and the target, the rows

Hops	Single Computer Payload 270Kb [MB/s]	Single Computer Payload 5Mb [MB/s]	Two Computers Payload 270Kb [MB/s]	Two Computers Payload 5Mb [MB/s]
1	15.44	15.62	13.54	13.29
2	5.78	4.93	4.57	2.26
3	2.57	2.05	2.23	**[Crash]**
4	1.94	1.34	1.88	**[Crash]**

(a) Yjs synchronization performance of the MinCon strategy without packaging.

1	15.36	15.64	13.51	13.32
2	10.43	9.23	8.32	7.85
3	6.59	5.31	4.78	4.16
4	4.53	3.8	3.84	3.59

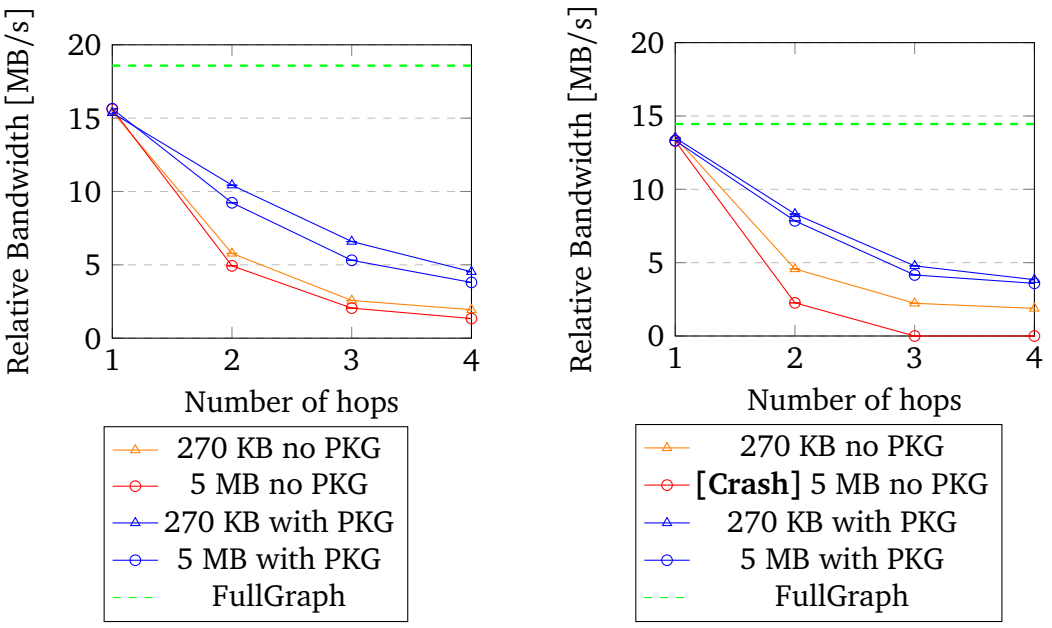
(b) Yjs synchronization performance of the MinCon strategy with packaging.

Table 9.6. Experimental values measured depending on the distance (hops) from peer `p0` based on the topologies shown in Figure 9.3. Peer `p0` starts the conversations across all peers.

showing the relative bandwidths for the 1, 2, and 3 hops would normally hold the same values as the 4 hops row, however in order to understand how the distance impacts the performance, we evaluated the bandwidths of those rows with additional experiments ran on their subset topologies. E.g, the 1 hop row is evaluated with the linear topology created by peers `p0-p1`, the 2 hops row is evaluated with the linear topology created by peers `p0-p1-p2`, and the 3 hops row is evaluated with the linear topology created by peers `p0-p1-p2-p3`. We also run the same experiments with message packaging enabled (see Table 9.6b) and then disabled (see Table 9.6a).

In Figure 9.4 we plot the values for an easier comparison of the results:

• In the single device experiment shown in Table 9.6), as expected, we see that the full-graph strategy performs better than any other strategy with an average relative bandwidth of 18.58 MB/s. The MinCon strategy behaves similarly for all 1 hop experiments, in fact when only two devices are connected, the packaging feature does not influence the Yjs synchronization at all. It is interesting to see that the MinCon strategy for the 1 hop experiment is on average 16.3% slower than the results obtained by full-graph strategy, meaning that the multiple queries to the LPC component in the peers create a substantial overhead in the synchronization process. From the the gathered data we can also derive the



(a) Comparison when peers are deployed on a single device (see Figure 9.3a).

(b) Comparison when peers are deployed on multiple devices (see Figure 9.3b).

Figure 9.4. Relative bandwidth comparison of the Yjs synchronization for the linear topologies shown in Figure 9.3 compared with the full-graph strategy. The evaluation is performed both with payloads of 270 KB and 5 MB. The full-graph strategy is displayed only for the average values of the 1 hop experiment, the line is then prolonged in order to help with the comparison with other strategies.

rates of bandwidth losses compared to the 1 hop experiment for each individual point in the plots: ⋆ in the 270 KB payload experiment without packaging we can see that the bandwidth for the 2 hops is on average equal to the 37.43% of the 1 hop experiment, the 3 hops is on average equal to 16.64%, and the 4 hops is on average equal to 12.56%; ⋆ in the 5 MB payload experiment without packaging the rates are lower and respectively equal on average to: 31.56%, 13.12%, and 8.58%; ⋆ in the 270 KB experiment with packaging we have the following rates: 67.9%, 42.9%, 29.9%; ⋆ in the 5 MB experiment with packaging we have the following rates: 59.01%, 33.95%, 24.29%. We can see that enabling the packaging feature in the MinCon strategy increases the speed of the Yjs synchronization of on average more than 100% for all experiments above the 1 hop distance. We can also see that the size of the message decreases the performance of the

synchronization process: the bigger the message, the slower is the relative bandwidth between the devices. This overhead is due to the time it takes for the LPC to receive the message from the buffer, unpack, read and forward it to another device.

• In the multi-device scenario shown in Table 9.6 we get similar results. The full-graph strategy is always faster and has a better performance than the MinCon strategy with an average bandwidth of 14.45 MB/s. The Yjs synchronization process crashed during the experiments with the 5 MB payload and no message packing starting from 3 hops. The peer throwing the first error was always `p1`, which was not able to handle the transmission of all the messages to all other peers without throwing a timeout error. Without message packaging, `p1` receives three 5 MB messages from `p0`, two of them must be forwarded to the other peers, while in the meantime it needs to patch its own liquid state and create new messages that must then be forwarded to all other peers. `p1` always crashed after the state was patched and before they ended the synchronization process, because the peers were never able to agree when the state was consistent on the topology. The bandwidth losses compared to 1 hop run are the following: ⋆ in the 270 KB experiment without packaging we have the following rates: 33.75%, 16.47%, 13.88%; ⋆ in the 270 KB experiment with packaging we have the following rates: 61.58%, 35.38%, 28.42%; ⋆ in the 5 MB experiment with packaging we have the following rates: 58.93%, 31.23%, 26.95%. While all the individual values in the multi-device experiments are lower when compared to the single device experiments, we can see that the rates follow similar trends.

In these experiments we can see that the size of the message has a slight impact on the Yjs synchronization. In the ideal scenario, where the packaging feature is enabled, the relative bandwidth is from 5 to 13% faster when the payload of the messages is 270 KB instead of 5 MB. We can also see that when the message is too big, the current state synchronization can have some problems agreeing on the consistency of the state deployed on distant machines.

9.1.2.3 Routing Table and Message Packaging in Hybrid Topologies

In this section we investigate the impact of the RT and message packaging features on the performance of the Yjs synchronization on the hybrid topology we presented in Figure 9.1. The payload of all messages contains a patch of 270 KB and `p0` is the peer that starts the conversation.

In Table 9.7 we show the aggregated results of all experiments. For each row, we run the experiment 5 times and in each run we update the state of a liquid property on `p0` 10 times with an interval of 30 seconds between an update and the

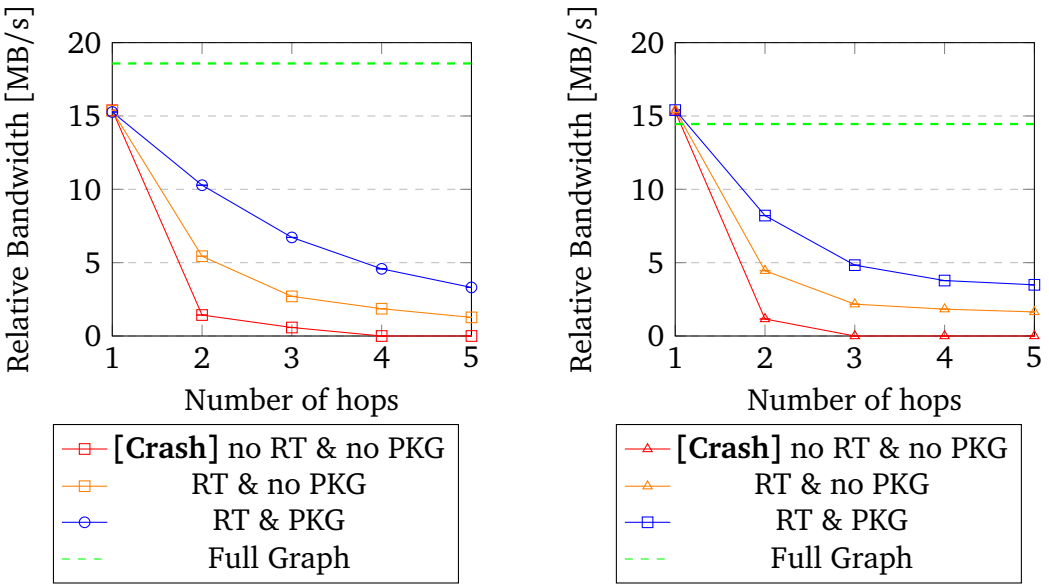
	Single Computer			Two Computers		
Hops	No RT no PKG [MB/s]	RT no PKG [MB/s]	RT PKG [MB/s]	No RT no PKG [MB/s]	RT no PKG [MB/s]	RT PKG [MB/s]
1	15.39	15.35	15.27	15.31	15.42	15.4
2	1.43	5.44	10.28	1.16	4.45	8.21
3	0.58	2.7	6.72	**[Crash]**	2.17	4.83
4	**[Crash]**	1.86	4.58	**[Crash]**	1.83	3.78
5	**[Crash]**	1.27	3.31	**[Crash]**	1.64	3.49

Table 9.7. Experimental values measured depending on the distance (hops) from peer `p0` based on the topologies shown in Figure 9.1. Peer `p0` starts the conversations across all peers.

other. The values displayed on the tables are the average of all relative upload bandwidth values computed on `p0`. Similarly to the experiments described in the previous section, the rows for the 1, 2, 3, and 4 hops are computed with additional experiments ran on the subset topologies consisting of all peers that have an equal or lower distance to the number of hops shown in the hops column.

In Figure 9.5 we plot the results of both the single and multi-device experiments: • in the single device experiments (shown in Figure 9.5a) we can see that lowering the number of messages exchanged between the peers with the RT and message packaging feature is necessary for the integrity of the state synchronization. In fact the synchronization crashes when both features are disabled starting from the 4 hops experiments. Similarly to the previous section, the conversation between the peers never finishes and Yjs throws an error caused by a timeout. We can also see the sudden drop in the performance starting from the 2 hops experiment, where the relative bandwidth is more than 10 time slower than the 1 hop experiment. The Yjs synchronization performs better when both RT and message packaging feature are enabled. The bandwidth losses compared to the 1 hop run are the following: ⋆ in the RT without packaging experiment we have the following rates: 35.44%, 17.59%, 12.11%, 8.27%; ⋆ in the RT with packaging experiment we have the following rates: 67.32%, 44.01%, 29.99%, 21.67%;

• in the multi-device experiment (shown in Figure 9.5b) we see that the experiment without RT and message packaging crashes even earlier than the single device experiment. The delay of communication between the devices influences the conversation, which starts crashing from the 3 hops experiment. The bandwidth losses compared to the 1 hop run are the following: ⋆ in the RT without packaging experiment we have the following rates: 28.86%, 14.07%, 11.86%,



(a) Relative bandwidth comparison with Yjs synchronization for the single device topology shown in Figure 9.1c.

(b) Relative bandwidth comparison with Yjs synchronization for the multiple devices topology shown in Figure 9.1d.

Figure 9.5. Relative bandwidth comparison of the Yjs synchronization with an image size of 270 KB for a topology with 9 peers with and without routing table and message packaging. The values of the full-graph strategy is displayed only for the 1 hop experiment and then the line is prolonged in order to help with the comparison with other strategies.

10.63%; ⋆ in the RT with packaging experiment we have the following rates: 53.31%, 31.36%, 24.54%, 22.66%;

The rates of the experiments when the RT is enabled are following the same trends we showed in the previous section when they were performed on the linear topologies. The shape of the topology does not directly affect the performance of the Yjs synchronization.

9.1.3 Discussion of the Results

The experiments we presented show that the full-graph strategy always outperforms the MinCon strategy. In fact, as expected, it has an higher relative average bandwidth in all experiments. However it is important to note that it is not always feasible to use the full-graph strategy on all deployments, because the

peers have access to a limited number of resources. The full-graph strategy is more demanding and while it has an higher relative bandwidth, it requires to allocate more memory on each peer when we compare it to the MinCon strategy. E.g, when `N` peers are participating in a topology, the full-graph strategy creates `N-1` communication channels on each of them, while the MinCon strategy creates `2*(N-1)` communication channels in total. Moreover, whenever a peer broadcasts an update to all participants, in the full-graph strategy it may be impossible to update every peer in real-time if peers have access to a limited bandwidth. E.g., in the full-graph strategy the peer in charge of broadcasting the update sends `N-1` times the same payload to all neighbour peers, while in the MinCon strategy, with RT and packaging enabled, the bandwidth usage is shared between peers.

This trade-off is important and must be taken into consideration when the developers select a strategy. When the peers are powerful and nothing limits their bandwidth, the developers can select the full-graph strategy. If the developers expect to connect slow peers with limited resources (e.g., IoT devices), they should consider to use the MinCon strategy, or create a strategy of their own. A possible strategy that developers can use for IoT devices can be an hybrid, where multiple IoT devices connect to a powerful node (e.g., an hub), and the hubs are then connected among each other.

The state synchronization tightly depends on the synchronization algorithm used. Currently we built the state synchronization with Yjs, assuming that all peers are masters and that any peer can change the liquid state at any time. In a master-slave environment the state synchronization process could be further improved and the conversations could be optimized in the MinCon strategy. In a master-slave environment, the peers would be required to check the consistency of their state only once against the version owned by the master, without checking the consistency with all other connected peers, consequently decreasing the total number of messages exchanged. In the general solution proposed by Liquid.js, the master-slave synchronization is not part of the core design decisions and is not currently available. In any case, the RT and message packaging features provided by the LPC do not depend on the current Yjs implementation, meaning that those features can be re-used for any kind of conversation and synchronization algorithm.

The payload size we sampled in the evaluation is also an overestimate of the average size of messages we expect to be exchanged between peers. The 270 KB and 5 MB payloads we used were *PNG* files meant to stress the synchronization process, however files are not the usual kind of data exchanged between peers in a Web application. Data is usually represented with objects in JSON format,

which do not necessarily change their whole content after the initial state migration. It is more common to change a single or multiple properties of an object instead of redefining it. With trickle synchronizations it is possible to synchronize only the delta that changed without sending the whole object to all peers. Yjs helps with lowering the amount of data exchanged between peers by automatically create payloads containing only the delta of updated objects.

9.2 Logic Layer

In order to study the feasibility and performance of the LWWs, in this section we present the results of an evaluation of the Liquid.js prototype implementation.

9.2.1 Test Scenario: Offloading Image Processing Tasks

The Liquid.js framework comes with various demo applications, including the liquid camera. This allows users to take pictures with their devices' Webcams, share pictures and display them across multiple devices, and apply a variety of image transformation filters. Applying filters to the images displayed on one device will immediately show the result on all copies of the image found across all connected devices. Since filtering images is a CPU-intensive operation, we have migrated the existing implementation based on WebWorkers to use the LWWPool. Figure 9.6 and 9.7 show the results of our preliminary experiments using LWWs.

9.2.2 Testbed Configuration

All experiments described hereafter are ran using different machines connected to the same private WiFi 5GHz network with the following hardware and OS specification: • **Laptop** (**L**): MacBook Pro (Retina, 15-inch, Mid 2014), 2.2 GHz Intel Core i7, macOS High Sierra Version 10.13.2, Chrome Version 64.0; • **Tablet** (**T**): Samsung Galaxy Tab A (2016), Octa Core 1.6 GHz, Android Version 7.0, Chrome Version 64.0; • **Phone** (**P**): Samsung J5 (2015), Quad Core 1.2 GHz, Android Version 5.1.1, Chrome Version 62.0.

In this study we show the performance for all shown configurations given the three different kind of devices. The policy loaded inside the LWW takes the decision not to or to offload the execution to other devices based on a predefined static configuration used to explore all possible device combinations in the experiments.

9.2.3 Workloads

In this evaluation we run two different experiments by applying various filters to the same picture. In the first "Edge Detection" experiment (see Figure 9.6a) we apply to the image the Sobel operator filter (using a 3x3 convolution matrix kernel). In the second "Improved Edge Detection" experiment (see Figure 9.6b) we improve the result of the edge detection by chaining multiple filters. Compared to the first, the second experiment puts a larger workload on the device

CPUs as they run multiple filters with larger kernels. The chained filters are: 1. a sharpening filter implemented by using a convolution filter with a 5x5 kernel; 2. an embossing filter using a 5x5 kernel; 3. the Sobel operator filter using a 5x5 kernel.

For each experiment we apply the filter on two different image resolutions, consequently changing the size of the message exchanged between devices. Both versions of the image are encoded using the *PNG* format and are transferred with messages of size **94196 bytes** and **198560 bytes**.

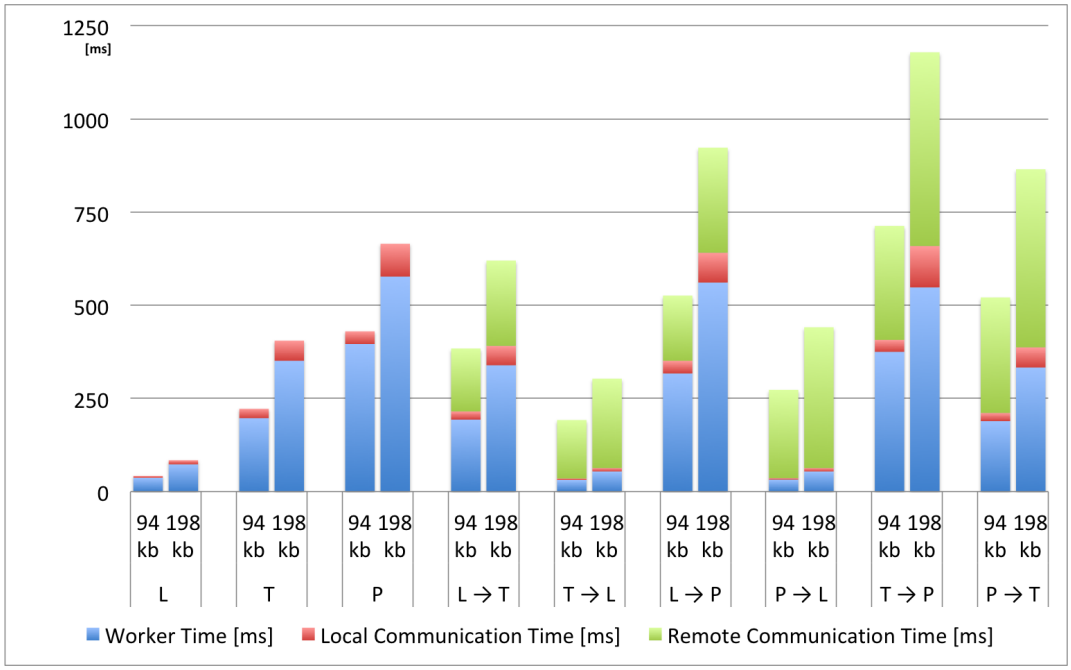
9.2.4 Measurements

We run each experiment 10 times, during each trial we applied the filters 25 times for both image sizes for all different device offloading combinations. Between two trials we reset the execution environment by restarting the Web browser on all devices. The values of the execution time shown in Figure 9.6 are computed as the average over the 10 trials.

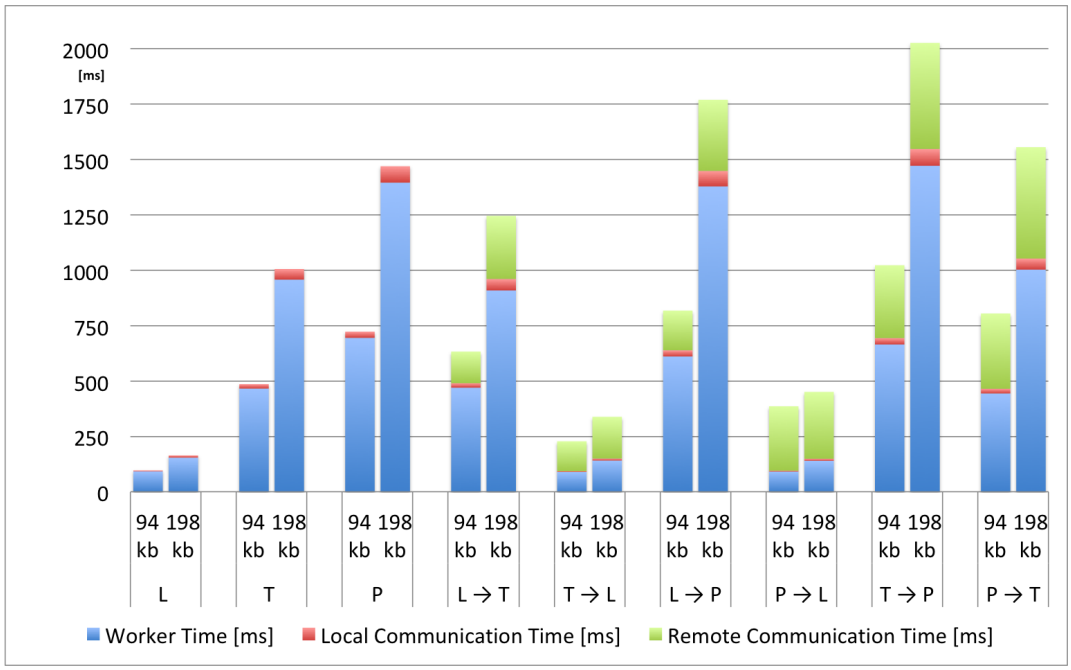
9.2.5 Results

The charts show the average time spent by the devices in order to execute a submitted task. Using three different colors we highlight the time elapsed during (see Equation 9.1): the *worker processing time* in blue, the *remote (or cross-device) communication time* in green, and the *local (or intra-device) communication time* in red. The *worker time* represents the time spent running the LWW script to process the submitted task; the *remote communication time* is spent during the transfer of the submitted task and its output result between the local and remote devices; the *local communication time* includes the time for sending and receiving back the task from the main thread to the LWW, the time employed for message marshalling and unmarshalling, the time spent idle in a message queue, and the overhead of the logging needed to gather performance data for this evaluation.

$$\begin{aligned} Process_{time}^{total} = PromisePreProcess_{time} + Send_{time}^{offload} + \\ MessageQueue_{time} + WorkerExecution_{time} + Marshalling_{time} + \\ Send_{time}^{response} + PromisePostProcess_{time} \end{aligned} \tag{9.1}$$

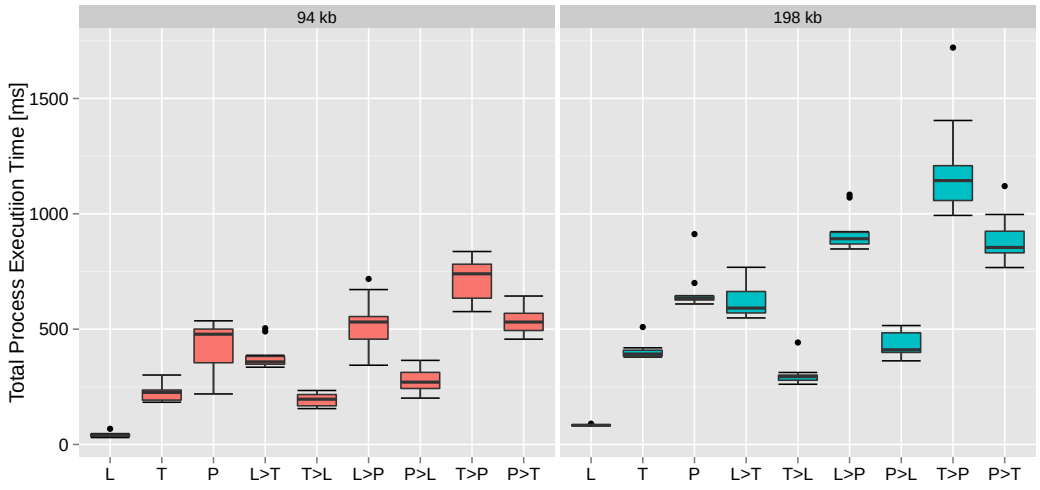


(a) Edge detection workload

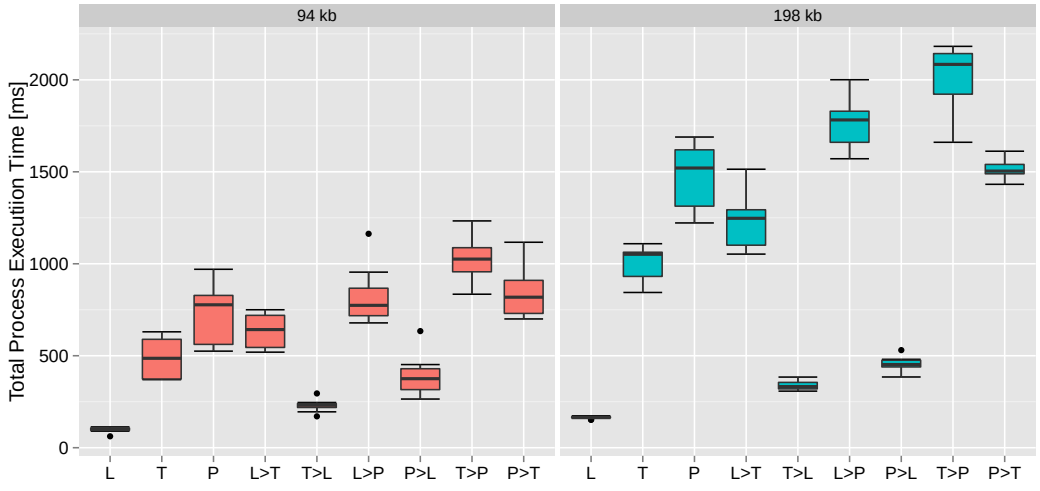


(b) Improved edge detection workload

Figure 9.6. Average processing and communication time of the LWW offloaded across different pairs of devices (L Laptop, T, Tablet, P Phone)



(a) Edge detection workload



(b) Improved edge detection workload

Figure 9.7. Boxplots of the total process execution time of the LWW offloaded across different pairs of devices (L Laptop, T, Tablet, P Phone)

9.2.5.1 Edge Detection Case

In Figure 9.6a and 9.7a the fastest execution happens on the laptop (**L**) without any offloading. The laptop finishes the process on average about five times faster than the tablet (**T**), and nine times faster than the phone (**P**) for both image sizes. It is interesting to see that every time the laptop was configured to offload work to any other device (**L→T**, and **L→P**), the overall execution took longer due to the slower *worker processing time* of the remote devices and the additional *remote communication time* required to transfer the task and the response between the devices; the same behavior can be observed when the tablet offloads its work to the phone (**T→P**).

In the **T→L** and **P→L** offloading configurations, the overall execution is faster when compared with the local execution without offloading cases. The elapsed *worker time* of the laptop is so low compared to the one of the tablet and the phone that, despite the penalty due to the remote communication time, the total execution time remains lower. **T→L** is on average 81% faster than **T** and **P→L** is on average 64% faster than **P**. Despite the expectation that also the configuration **P→T** would execute faster than **P**, this was not observed because of the *communication time*. So there were no benefits in offloading the task from the phone to the tablet, in fact in this case the performance worsened.

As a side note, we observed that the WiFi data transmission performance depends on the device, with the phone's available bandwidth being smaller than on the other devices. This behavior is evident when comparing all offloading configurations where the phone is involved with all other configurations. In particular the communication time between the phone and the tablet is double than the time between the laptop and the tablet. This could also be caused by the physical proximity of the devices during the tests which may have led to some interference as indicated by changes of the WiFi signal strength on the devices. We did not attempt to shield the devices to reduce measurement noise because our goal was to reproduce real-world usage conditions.

From this experiment we can conclude that it is possible to benefit from using LWWs and thus it is possible to lower the total processing time by offloading tasks to nearby devices. However, this can be achieved only when the extra communication overhead is smaller than the gained processing time due to the faster remote CPU.

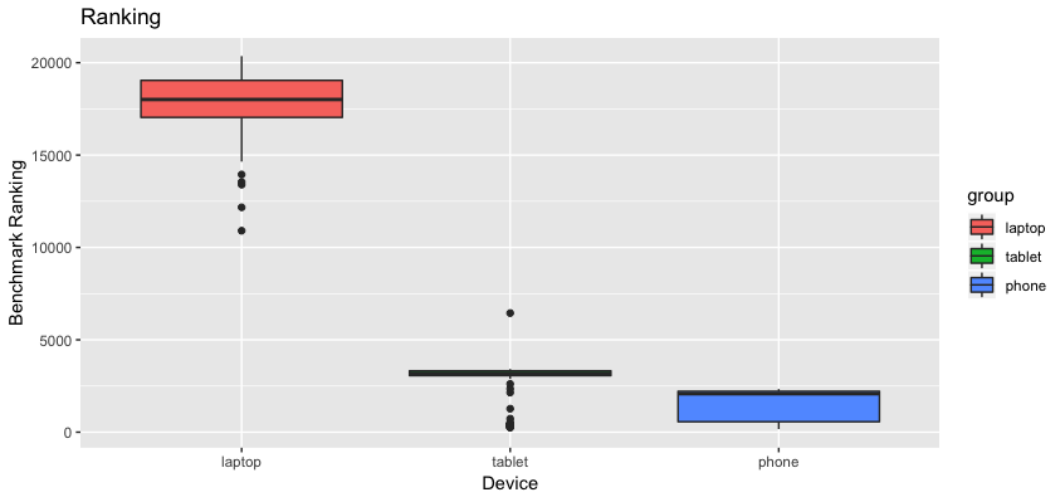


Figure 9.8. Boxplot of the benchmark scores for each device.

9.2.5.2 Improved Edge Detection Case

In Figure 9.6b and 9.7b we stress the devices more as we increase the workload exerted on the LWWs. On average the *worker processing time* for this experiment is 248% longer on all devices when compared to the previous experiment. We can observe that the *local communication time* is unaffected by the experiment, but the average *remote communication time* slightly changes due to the previously discussed noisy WiFi channel.

Offloading computations to the phone never registers lower process execution times (**L→P** and **T→P**), which is the conclusion we observed before.

Particularly interesting in the second experiment are the values registered in configuration **P→T** compared to values registered in **P**. In this case we observe that again on average **P** is slightly faster (82-85ms difference) than **P→T**. Still, if we examine the trend by including the data from the experiment before we can see that the longer the *worker time*, the better it is to offload workload from **P** to **T**. Eventually, for heavy workloads, offloading to a tablet would be better than executing the tasks on the phone, because the *remote communication time* remains mostly constant for the same image size while the *worker time* constitutes the dominant factor.

9.2.6 Micro-Benchmark evaluation

Can the micro benchmarking score accurately predict the capabilities of a connected device? We answer this question by comparing the scores returned by the

Table 9.8. Average benchmark ranking and average processing.

	94kb				198kb					
	Comb		Sobel		Comb		Sobel		Benchmark	
	[ms]	%	[ms]	%	[ms]	%	[ms]	%	Score	$\%^{-1}$
laptop	91	19.5	33	16.8	145	15.2	60	17.3	17898	15.1
tablet	466		196		953		345		2707	
laptop	91	13.7	33	9.1	145	10.3	60	10.5	17898	8.7
phone	662		363		1413		569		1550	
tablet	466	70.4	196	54.0	953	67.4	345	60.6	2707	57.3
phone	662		363		1413		569		1550	

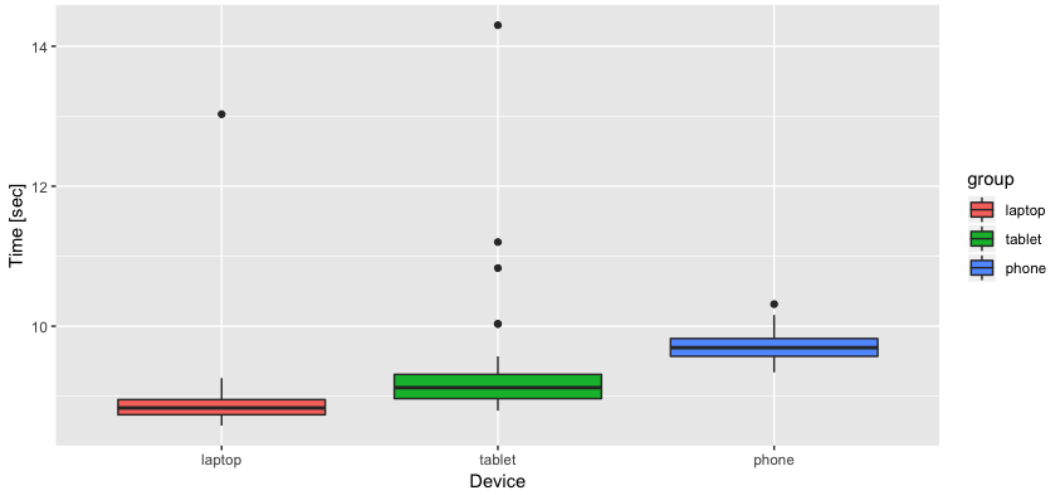


Figure 9.9. Boxplot of the benchmark execution times.

test benchmark against the results obtained in the previous sections.

Figure 9.8 shows the results obtained by running the benchmark a total of 200 times for each device. The benchmark is executed at startup and then it is repeated periodically every *300 seconds*. The application is restarted after it has completed 50 benchmarks, meaning that the application runs continuously for 15000 seconds (approximately 4 hours), before the device is restarted. In order to reduce the measurements noise, during the benchmark execution the user does not interact with the device, simulating a comparable scenario with the previous evaluation.

As expected the score for the laptop is higher than the other devices, with an average score of *17898*, while the tablet scores *2707.3* and the phone *1550.4*.

In Table 9.8 we compare the average worker execution times against the

benchmark scores. We list all possible pairs of devices and compute both the ratio between their respective average worker execution times and their average score returned by the benchmark. In *yellow*, *orange*, and *red* we highlight the average ratios computed for the same couple of devices. Since the machines do not change, we expect that the execution ratios do not change within the same pair even if the experiment is different. Whenever the LWW executes a longer task on a machine, then we expect it proportionally increases also on the other one. In all three couples, we see that the average ratio between the sampled benchmarked ratio and the real world example are similar. The benchmarked ratio for *taptop-tablet* differs on average the *13%* of the real world scenario ratio, the *laptop-phone* ratio differs on average the *20%* from the real world ratio, and the *tablet-phone* ratio only *9%*.

Figure 9.9 shows how much time it takes to execute the benchmark on each device. The execution time is stable, with very few outliers on the tablet device. On average, between all three the devices, it takes *9.3 seconds* to execute the micro-benchmark.

9.3 View Layer

We show the expressiveness of liquid media queries by designing the *liquid-style* components on a realistic multi-device video player application.

The video player is built with four components (see Figure 9.10): • the **video** component which displays and plays the video; • the **video controller** component which allows the user to play/pause and seek to a specific time in the selected video; • the **suggested videos** component that displays a list of recommended videos, which can be selected to be played; • the **comments** component where the user can read or post comments about the video.

These components can be deployed across different devices (phones, tablets, laptops, and televisions) owned by one or multiple users.

It is best to display the video component (see Listing 9.1) on the devices with big screens, for this reason we define three liquid media query expressions including the attributes `device-type: laptop`, `device-role: display`, and `device-ownership: shared` with different priorities. The rule for `device-type: laptop` has an higher priority over the rule defined for the *comments* component (see Listing 9.2) so that whenever a laptop device is available, the video component is migrated to the laptop. If the user configures the role of any device and assigns the role *display* to it, then this device will have priority over the laptop. Finally, if there are multiple users connected to the application (attribute *min-users:2*), the priority for deploying the *video* component is given to *shared* devices (e.g., a television).

The video controller component (see Listing 9.3) defines a liquid media query expression with the attribute *clone:1-user*. The clone rule migrates the component to a phone owned by a user, then it clones the component for every other user, if they connect at least another phone to the application.

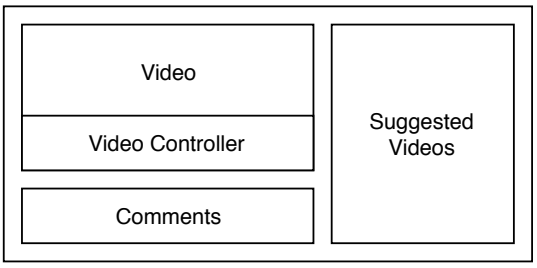


Figure 9.10. Liquid video player UI split into four components: video, video controller, suggested videos, comments

Listing 9.1. The liquid-style elements defined for the video component.

```
<liquid-style device-ownership="shared" min-users="2" priority="4">
  <!-- CSS Style Sheet 1 -->
</liquid-style>

<liquid-style device-role="display" priority="3">
  <!-- CSS Style Sheet 2 -->
</liquid-style>

<liquid-style device-type="laptop" priority="2">
  <!-- CSS Style Sheet 3 -->
</liquid-style>
```

Listing 9.2. The liquid-style element defined for the comments component.

```
<liquid-style device-type="laptop">
  <!-- CSS Style Sheet 1 -->
</liquid-style>
```

Listing 9.3. The liquid-style element defined for the video controller component.

```
<liquid-style device-type="phone" priority="2"
    clone="1-user">
    <!-- CSS Style Sheet 1 -->
</liquid-style>
```

Listing 9.4. The liquid-style elements defined for the suggested videos component.

```
<liquid-style device-type="phone">
  <!-- CSS Style Sheet 1 -->
</liquid-style>

<liquid-style device-type="tablet" priority="3">
  <!-- CSS Style Sheet 2 -->
</liquid-style>
```

The suggested video component (see Listing 9.4) defines two styles: one for tablets and the other for phones. The tablet style has an higher priority with respect to the phone style.

9.3.1 Scenario 1: Second User Connects a Smartphone

In Figure 9.11 we show the component redistribution for a set of devices before and after a second user connects to the application. The initial configuration with only devices owned by *UserA* is obtained following the priorities associated with the liquid-style elements of each component. Starting from the suggested video component, which migrates to the tablet, then the video component migrates to a laptop device, because the higher priority rules it holds are not accepted by any other device. The video controller migrates to a phone device, but it is not cloned on both available phones because of the clone rule set to `1-user`. Finally, the comments component migrates to the second laptop device.

After *UserB* logs in the application and connects an additional phone device, the UI is redistributed as follows. The video component is migrated to the television device because of the *ownership* and *min-users* rules have now higher priority 4. The video controller component is cloned to *UserB*'s phone.

9.3.2 Scenario 2: Dynamic Device-Role Change

In Figure 9.12 we show an example of dynamic change in the metadata configuration of the connected devices. The initial device configuration is not accepted by at least one liquid media query defined in the video controller component, and the target device for the video and comments components points the same laptop. Starting from the highest priority, the suggested video component is deployed on the tablet and the video component is deployed on the laptop. Since the laptop component is already the target of the video component, the comments component migrates to the television, which was ranked as the next possible target for migration. The video controller component is deployed on the tablet device with the lowest priority.

When *UserA* assigns the role *display* to the television, the device metadata changes. The UI is redistributed and the video component migrates to the television, because the *liquid-style* that defines the property *device-role* is now accepted by the device with an higher priority. The comment component migrates to the now available laptop device.

	Component	Video	Video Controller	Comments	Suggested Videos
Initial Configuration	*Priority*	2	2	1	3
	Migration Target	laptop 1	phone 1	laptop 2	tablet
User B connects phone	*Priority*	4	2	1	3
	Migration Target	**television**	(phone 1)	(laptop 2)	(tablet)
	Cloning Target		**phone 2**		

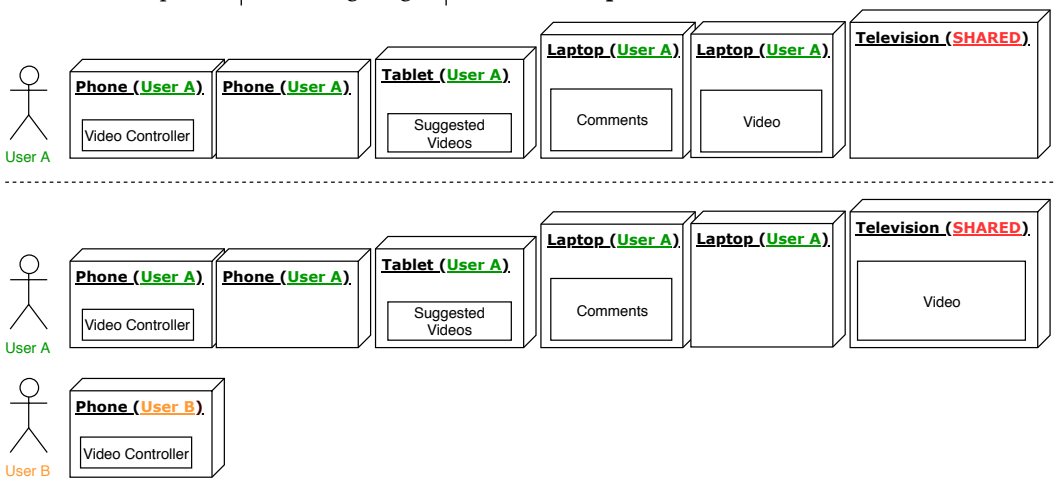


Figure 9.11. When a second user connects to the application the video component is migrated to the shared device and a new instance of the video controller is deployed on the new user's phone.

	Component	Video	Video Controller	Comments	Suggested Videos
Initial Configuration	*Priority*	2	0	1 (0)	3
	Migration Target	laptop	tablet	television	tablet
Television role set to display	*Priority*	4	0	1	3
	Migration Target	**television**	(tablet)	**laptop**	(tablet)

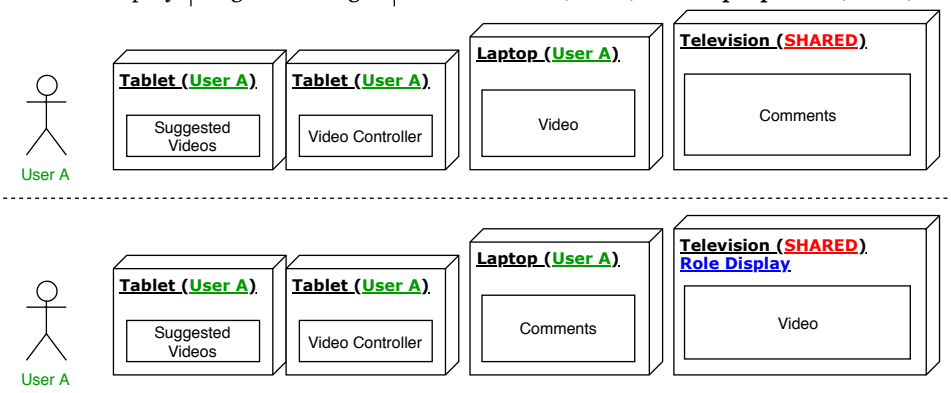


Figure 9.12. After the television device changes role configuration, the video and comments components are swapped following different priorities.

9.4 Building Liquid Web Applications with Liquid.js

In this section we discuss how to build liquid components in Liquid.js. The following sections shows the following five examples: 1. in the first example we present all the necessary steps for trasforming any standard Polymer component into a liquid component; 2. in the second example we show how to build a liquid component with multiple liquid properties with the liquid Googlemap component; 3. in the third example we present how to build a simplified version of the liquid Youtube player using containers; 4. in the fourth example we discuss liquid UI wrappers and present our manual position-aware migration UI wrapper, that allows users to use liquid primitives with drag-and-drop gestures; 5. in the last example we present an advanced experiment built with Liquid.js, in which we show that we can code components on the Web browsers and deploy them on other connected clients without the need of a Web server.

9.4.1 Converting Standard Polymer Components into Liquid Components

It is possible to convert any Polymer component uploaded in the WebComponents Catalogue [Web20] into a liquid component by following three steps. In particular in this section we decided to show the conversion of the `<paper-input>` component [The18c], one of the basic Polymer elements that encapsulate the behavior of an `<input>` element with the standard appearance (CSS styles) of the *paper-ui* element suite. The three steps are the following:

1. In this step we setup the new Polymer component we are going to make liquid. First we create a new file that contains the definition of the liquid component, we name this file `liquid-polymer-input.html`. Inside this file we write the required and standard minimum Polymer component template as shown in Listing 9.5. In this template we are defining a new component labeled `liquid-component-example`, which contains: • a `<template>` element (line 3) in which we will append the HTML that will be loaded inside every instance of our `<liquid-component-example>`; • a `<script>` element (line 6) in which we will define the logic of the component with a JS snippet. In the standard Polymer template, the `<liquid-component-example>` does not import any behavior yet (line 9), and has no properties (line 10). This template is not Liquid.js specific and can be used to create any Polymer component even without liquid features. In order to create an instance of the `<paper-input>` component, we then download the source of the `<paper-input>` source from the Web components catalogue [The18c] into the assets folder of your Web application. Once the compo-

Listing 9.5. Building liquid components step 1: default Polymer template

```
1  <!-- Imports here -->
2  <dom-module id="liquid-component-example">
3    <template>
4      <!-- HTML here -->
5    </template>
6    <script>
7      Polymer({
8        is: 'liquid-component-example',
9        behaviors: [],
10       properties: {},
11
12       // Methods here
13     });
14   </script>
15 </dom-module>
```

Listing 9.6. Building liquid components step 1: HTML import definition

```
1 <link rel="import" href="/paper-input/paper-input.html">
```

nent can be accessed from the Web server, we can preprend the HTML import shown in Listing 9.6 before our component definition. We suggest to import it on top of the `liquid-polymer-input.html` file at line 1, however the developers can choose to import it anywhere else in their Web application, e.g., they can globally import it in the main *.html* file where the Liquid.js framework is initialized. The best practice is to import the `<paper-input>` inside the files that depend on it, since the Web browser and Liquid.js will download it only when necessary.
Once the `<paper-input>` is imported into our component, we can finally access and instantiate it inside the `<template>` element. We append it at line 4 and then we assign a string to the property `value`, e.g., `"Insert Text Here"`, as shown in Listing 9.7.

2. Now that the component is ready, we make it liquid by injecting the `LiquidBehavior` into the component behaviors array. In Listing 9.8 we show how line 9 of our component definition changes with the newly injected behavior. The developers do not need to import the behavior on top of the file, because it is automatically imported by the Liquid.js framework upon initialization, and consequentially the object `LiquidBehavior` can be accessed globally anywhere in the Web application.

Listing 9.7. Building liquid components step 3: append the `<paper-input>` element

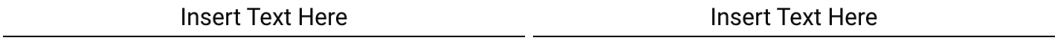
```
1 <paper-input value="Insert Text Here"></paper-input>
```

Listing 9.8. Building liquid components step 2: inject the `LiquidBehavior`

```
1 behaviors: [LiquidBehavior],
```

With the injected behavior, the component is now liquid and can fully access all the features provided by the Liquid.js framework. Any instance of the `liquid-component-example` can now be migrated, forked or cloned to any device, even if we did not define any liquid property yet. Without liquid properties the component does not define its own liquid state, thus when the component moves among the devices, it does not bring any change to the state with it. E.g., if the users change the value of the `<paper-input>` inside the liquid component and then migrate it to another device, the the new value will not migrate with the component, and once the migration finishes, the value will be set to the default `"Insert Text Here"`.
In Figure 9.13 we show how the Web browser displays the `<paper-input>` component (Figure 9.13a) and our new `<polymer-component example>` (Figure 9.13b). As it is possible to see, the components are visually the same and both can be used as inputs in a solid application, since they both define the same `<input>` behavior.

3. In the last step we define a new liquid property and bind it to the attribute `value` of the `<paper-input>` element. This can be done by adding a new property inside the definition of our liquid component at line 10. In Listing 9.9 we show how to create a property labeled `myProp` of type String, that has a default value and is annotated as `liquid`. We can then bind `myProp` to the attribute `value` as shown in Listing 9.10. It is important to note that property binding is a core feature of Polymer and is not Liquid.js dependant, moreover the liquid properties can define any other standard attribute that could be defined in a standard Polymer property, e.g., observers. The type is not required, however it



(a) `<paper-input>` (b) `<liquid-component-example>`

Figure 9.13. Standard Polymer component versus liquid component

Listing 9.9. Building liquid components step 3: define a liquid property

```
myProp: {
    type: String,               // Optional
    value: "Insert Text Here", // Default value
    liquid: true                // Makes a property liqiud
}
```

Listing 9.10. Building liquid components step 3: property binding

```
<paper-input value="{{my-prop}}"></paper-input>
```

helps Liquid.js to initialize an optimized proxy trap when the developers define it in advance.
Now that the component contains at least one liquid properties, we can finally migrate, fork and clone it to a target device and it will bring the new updated `value` when it moves. Components can contain any number of properties and each property can either be liquid or non-liquid.

The final snippet of code for the `liquid-component-example` is shown in Listing 9.11.

Now that we built our component we can load it in our liquid Web application. We can load a liquid component in two different ways: • we can instantiate it programmatically after the Liquid.js is initialized; • or we can write it inside the `.html` file of the Web application as if it is a HTML element.

In the first case, the developers can use Promises [Moz20c] as shown in Listing 9.12. The Liquid object is accessible globally and must be configured as we discussed in Section 8.1. Once Liquid.js finishes the configuration process, the following three Promises' chains can be used to instantiate the `<liquid-component-example>` (all chains produce the same final result):

1. in the first chain the developers load the liquid component type first and then instantiate the component into the Web application. Since the developers do not specify a target for the instantiation, the component is created on the local machine and appended to the `<body>` element;

2. in the second chain the developers use a helper function that loads and creates a new component with a single operation. The developers do not have to worry to load components multiple times, because when Liquid.js detects that a component is already loaded and stored in the local database, it will skip the operation;

3. in the last chain the developers instantiate the component by passing two additional parameters: an HTML element and an object. The first parameter is

Listing 9.11. Building liquid components final snippet

```
<link rel="import" href="/paper-input/paper-input.html">
<dom-module id="liquid-component-example">
  <template>
    <paper-input value="{{my-prop}}"></paper-input>
  </template>
  <script>
    Polymer({
      is: 'liquid-component-example',
      behaviors: [LiquidBehavior],
      properties: {
        myProp: {
          type: String, value: "Insert Text Here", liquid: true
      }},
    });
  </script>
</dom-module>
```

the target local element where the liquid component will be appended, e.g., in this example the `<body>` element, the second parameter defines the options of the *create* operation. In this particular case it allows the developers to specify if a liquid UI wrapper should be loaded with the component, e.g., in this case they do not load it (the default value of the `liquidui` attribute is `false`).

If the developers do not want to instantiate components programmatically, they can instantiate the component directly in the HTML of the Web application as shown in Listing 9.13. During the configuration process, after the framework downloaded the needed dependencies from the Web server, it parses the whole DOM tree of the Web application and searches for the definition of liquid components. When a liquid component is detected, Liquid.js automatically loads the component type and instantiates the components in the position where it finds the definitions. The developers can also pass options as attributes of the element they are creating, e.g., in this example the `liquidui` attribute is set to `false`.

Listing 9.12. Loading liquid components programmatically with Promises

```
// Using the Liquid.js API
Liquid.configure(liquidOptions)
    .then(function(){
        Liquid.loadComponentType('component-example')
    })
    .then(function(){
        Liquid.createComponent('component-example')
    })

// Using an helper function of the Liquid.js API
Liquid.configure(liquidOptions)
    .then(function(){
        Liquid.loadAndCreateComponent('component-example')
    })

// Using optional parameters
Liquid.configure(liquidOptions)
    .then(function(){
        Liquid.loadAndCreateComponent('component-example', document.
            querySelector('body'), {liquidui: false})
    })
```

Listing 9.13. Loading liquid components in the main `.html` file of the Web application

```
<html>
  <head> <!-- Headers --> </head>
  <body>

    <liquid-component-example liquidui="false">
    </liquid-component-example>


    <script src="/liquidAPI.js"></script>

    <script>
      Liquid.configure()
    </script>
  </body>
</html>
```

9.4.2 Multiple Properties with the Liquid Googlemap Component

In this section we show a more complex example in which we instantiate multiple components with multiple properties for creating a multi-device map application. We reuse the `liquid-component-example` we built in the previous example (Section 9.4) and rename it to `liquid-component-input` in order to enhance the readability of the following listings. The map is drawn with the `google-map` [Goo16b] and the `google-map-directions` [Goo16c] components which can be found in the Web components catalogue. In Figure 9.14 we show how the `<liquid-component-googlemap>` we built looks when it is instantiated on a Web browser, and in Listing 9.14 we show the definition of the component. From Listing 9.14 we can point out the following features:

- The component defines five liquid properties: the `latitude` (line 19), the `longitude` (line 20), the `zoom` (line 21), the `startLoc` (line 22), and the `endLoc` (line 23). The first three properties are used to control the center position of the map, while the other two are used for drawing a path between two locations.
- The component imports the `google-map` (line 1) and the `google-map-directions` (line 2) components, which are then instantiated at lines 9 and 11. In the google-map component we bind the latitude, longitude and zoom properties, while in the google-map-directions component we bind the startLoc and endLoc properties. Whenever those properties are updated, the changes will be displayed on the map.
- Inside the component we also instantiate two instances of the `liquid-component-text` component (lines 5 and 7). These textual inputs are used by the users for writing the two locations that will be displayed on the map. Both components are loaded with the default liquid UI wrapper and define a

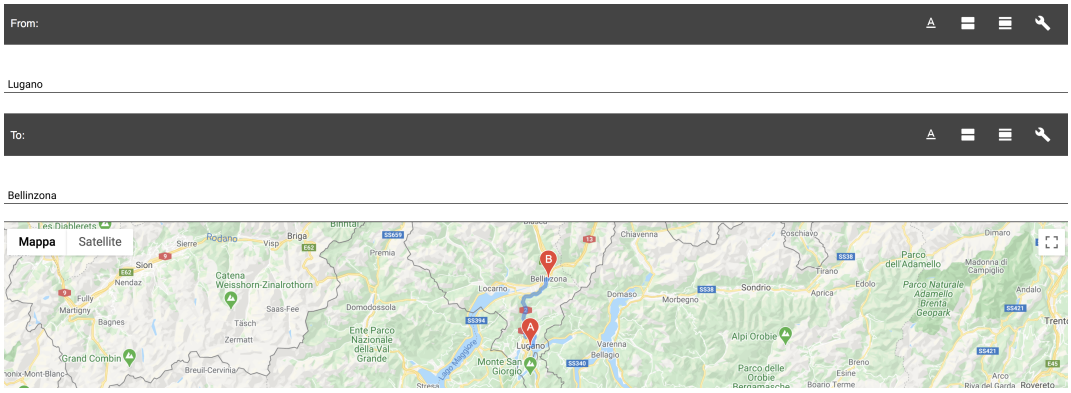


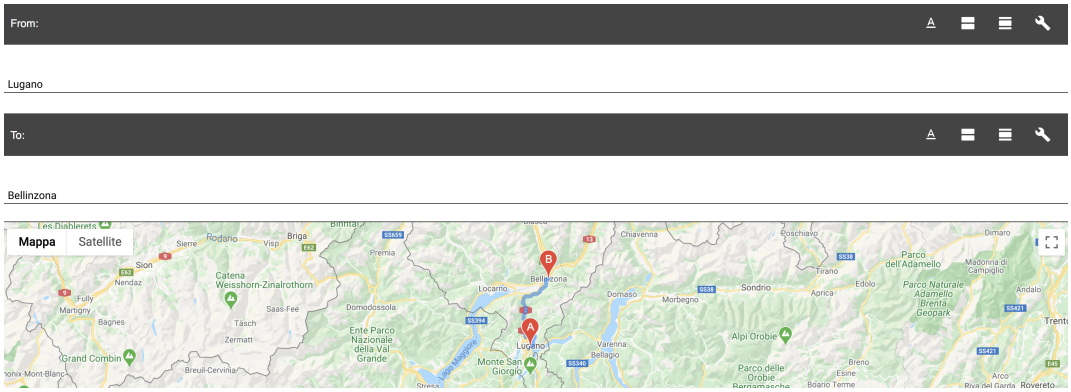
Figure 9.14. `<liquid-component-googlemap>` on the Web browser

Listing 9.14. Definition of the `<liquid-component-googlemap>` component

```
<link rel="import" href="/google-map.html">
<link rel="import" href="/google-map-directions.html">
<dom-module id="liquid-component-googlemap">
    <template>
    <liquid-component-text liquidui="default" liquidname="From:"
      value="{{start-loc}}"> </liquid-component-text>
    <liquid-component-text liquidui="default" liquidname="To:"
      value="{{end-loc}}"> </liquid-component-text>
    <google-map api-key="..." map="{{map}}" latitude="{{latitude}}"
      longitude="{{longitude}}" zoom="{{zoom}}"> </google-map>
    <google-map-directions api-key="..." map="{{map}}"
        start-address="{{start-loc}}"
      end-address="{{end-loc}}"> </google-map-directions>
    </template>
    <script>
      Polymer({
        is: 'liquid-component-googlemap',
          behaviors: [LiquidBehavior],
          properties: {
          latitude: { value: 37.77493, liquid: true },
          longitude: {value: -122.41942, liquid: true },
          zoom: { value: 15, liquid: true },
          startLoc: { value: "", liquid: true },
          endLoc: { value: "", liquid: true }
        }
      });
    </script>
</dom-module>
```

name that helps the users distinguish them. The given name is displayed on the liquid UI wrapper as can be seen in Figure 9.14.

Once the components are instantiated on a device, the users (or the developers) can migrate, fork, or clone them on multiple target devices. In all the following examples, since we are not defining a liquid container yet (we will show them in Section 9.4.3), the users need to clone each component individually. This means that when the users clone the `<liquid-component-googlemap>`, the component is cloned on the target devices, but it will not bring the `<liquid-component-text>` components with it. As discussed in the implementation of Liquid.js, when the developers intend to migrate/fork/clone the children of a liquid component together with the parent, they should define a liquid



(a) Desktop view

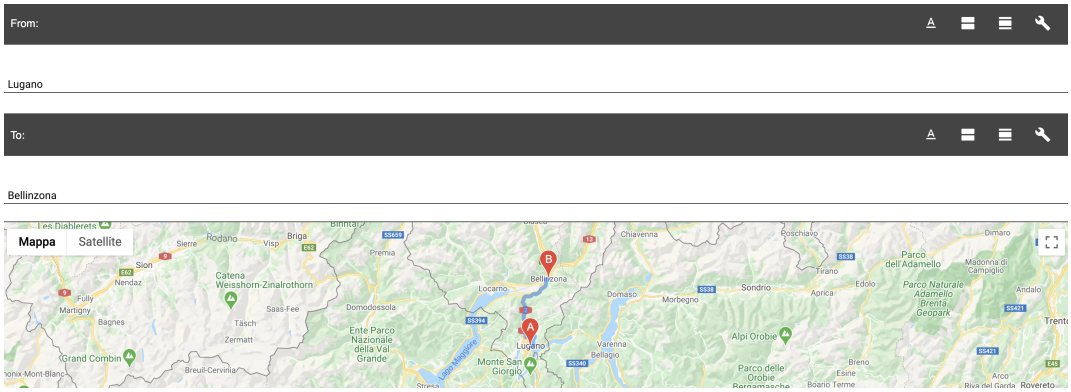
(b) Smartphone view

Figure 9.15. Multi-device deployment of `<liquid-component-googlemap>` component: the map is cloned on both devices

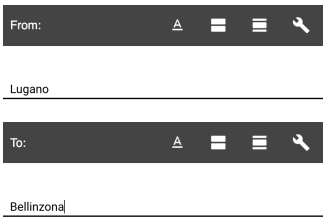
container.

In Figure 9.15 we show the view of the map application when the users decide to clone the map component from a desktop computer to a smartphone. The component cloned on the smartphone can adapt its size to the smaller display, however, even if the view port is smaller, we can see that the two maps are still sharing the same properties' values. The `latitude`, `longitude`, and `zoom` are synchronized between the devices and whenever they change, Liquid.js propagates the updates accordingly. Both devices are masters, meaning that both the desktop and the smartphone can change the value of the liquid state.

In Figure 9.16 we show another possible deployment for the same map application. The map is initially instantiated on the desktop computer, then the users clone the two input components on the smartphone device. In this scenario the smartphone is used as a companion device, in fact it owns a fraction of the whole Web application. In this particular case the smartphone is used as a controller for the map, which represents the view of the application and is deployed on a machine with a bigger screen. This example simulates the GPS device that can be found in a car, where the drivers can see the map on the dashboard in front



(a) Desktop view

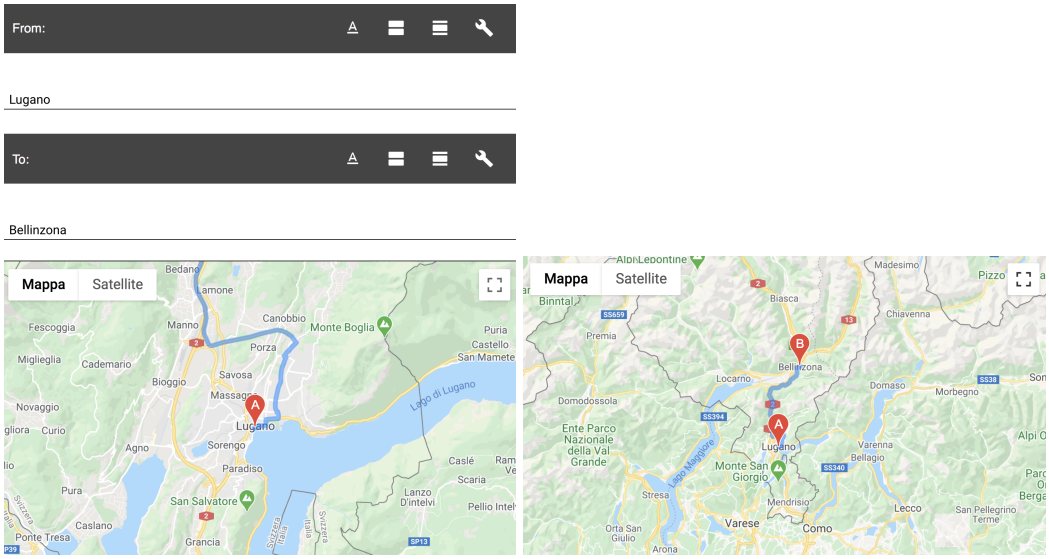


(b) Smartphone view

Figure 9.16. Multi-device deployment of `<liquid-component-googlemap>` component: the inputs are cloned on both devices

of them, while passengers can change the addresses directly from their phones. The drivers and the passengers can decide to search for locations from the GPS and display a path on the map, or they can search for new locations from their smartphone.

When the value of the input changes on either of the two devices, the value is propagated to all other components. The input values are updated in real-time, meaning that each letter written or deleted in an input, is displayed immediately in the other. While the `startLoc` and `endLoc` properties are synchronized between the desktop and the smartphone, it is interesting and important to note that the phone does not hold the values for the `latitude`, `longitude`, and `zoom` properties in this scenario. In fact the two cloned input components do not internally define those three liquid properties, and thus the desktop never propagates their state to the smartphone. During the lifespan of map application, the smartphone never receive the state of the `latitude`, `longitude`, and `zoom` properties as long as the map componented is not cloned on the smartphone. The smartphone would receive the values of those properties only if the map component



(a) Tablet 1 (b) Tablet 2

Figure 9.17. Multi-device deployment of `<liquid-component-googlemap>` component: `latitude`, `longitude`, and `zoom` are not liquid

itself would be migrated, forked, or cloned from the desktop computer into it.

In Figure 9.17 we show what happens when we slightly change the definition of the `<liquid-component-googlemap>` and deploy it on two tablets. In this example we change lines 19, 20, and 21 of Listing 9.14 and set the properties `liquid` to false. This means that the liquid state of the component changed, and the values of `latitude`, `longitude`, and `zoom` are not synchronized anymore among the two devices. When the users clone the map application from the first tablet to the second, Liquid.js synchronizes the new liquid state on the second tablet. Since the liquid state is now different, the second tablet can now navigate the map independently from what is viewed on the first tablet. The starting and ending address however are still displayed on both devices.

9.4.3 Liquid Containers with the Liquid Youtube Component

In the previous section we presented an application composed by multiple liquid components that moved independently from each other even if they were defined as children of a parent liquid component. In some scenarios this is the expected behavior when the users invoke the migrate, fork, and clone LUE primitives, but in other cases they expect to invoke those methods on both the parent and children at the same time, rather than invoking the primitives individually on each of them. Web applications already exploit the hierarchical DOM in order to understand when elements are children of another element, and thus we can reuse the same hierarchical system in order to understand which components are subordinated to a parent component. In such a way, Liquid.js can infer the children of the component passed as parameter of the LUE primitive and extend the primitive to them. In order to enable this feature, the developers need to annotate which liquid components are meant to be *containers* in the application. In Section 9.4.2 we presented the `<liquid-component-googlemap>` that potentially could be defined as a liquid container, since it internally instantiates two liquid components, however the developers designed it in such a way that only single parts of the component could be moved around.

In this section we present the `<liquid-component-youtube>`, that allows the users to load a video from Youtube. This component internally instantiates the `<liquid-component-slider>`, that can be used to control the feed of the video. When the youtube component moves from a device to another, it also brings the slider togheter with it.

In Listing 9.15 we show the definition of the `<liquid-component-youtube>`. This component uses the `<google-youtube>` [Goo17c] in order to load a video from Youtube. The component is imported at line 1 and loaded at line 4. We define four liquid properties bound to the `<google-youtube>`:

- `videoID` (line 17) is the identifier of the video that the users load. In this example the video is loaded with a defualt value, but the developers can add a `<paper-input>` in the `<template>` if they want the users to dynamically change the identifier at runtime;
- `state` (line 18) defines the current state of the video player, e.g., `state=1` means that the video is playing, `state=2` means that the video is currently paused. This property is bound to an observer labeled as `_stateChange` (definition at line 24) which is called every time the value of the property is updated. This method checks the `new` value of the state: if `state=1`, it will start the playback of the video, otherwise when the `state=2` it will stop it;
- `currenttime` (line 19) is the current playback time in seconds of the video.

Listing 9.15. Definition of the `<liquid-component-youtube>` component

```
1  <link rel="import" href="/google-youtube.html">
2  <dom-module id="liquid-component-google-youtube">
3    <template>
4      <google-youtube id="myVideo" video-id="{{videoID}}"
5        height="320px" width="600px" chromeless="0"
6        state="{{state}}" on-google-youtube-ready="_playerReady"
7        currenttime="{{currenttime}}" duration="{{duration}}">
8      </google-youtube>
9      <liquid-component-slider liquidui="default"
10       currenttime="{{currenttime}}" duration="{{duration}}">
11     </liquid-component-slider>
12   </template>
13   <script> Polymer({
14     is: 'liquid-component-google-youtube',
15     behaviors: [LiquidBehavior, LiquidContainerBehavior],
16     properties: {
17       videoID: { value: '848QV4H0IDE', liquid: true },
18       state: { value: 0, liquid: true, observer: "_stateChange" },
19       currenttime: { liquid: true, observer: "_timeChange" },
20       duration: { liquid: true }
21     },
22     ready: function () { Liquid.loadComponentType('slider'); },
23     _playerReady: function(e) { ... },
24     _stateChange: function(state) { ... },
25     _timeChange: function(time, prevTime) { ... },
26   }); </script>
27 </dom-module>
```

This liquid property is bound to the observer `_timeChange` (definition at line 24) which is called when the value of the property is updated. The method can check both the previous and new value of the `currenttime` and it can decide to skip the video to the specified number of seconds;

- `duration` (line 20) is the maximum video duration in seconds.

The video player is also attached to an helper function labeled as `_playerReady` (line 23), this method allows to detect when the video is loaded and takes care of starting it as soon as the method is called by the `<google-youtube>` component. At line 9 we instantiate the `<liquid-component-slider>` which is bound to the properties `currenttime` and `duration`. When the users move the slider, the new value of the property

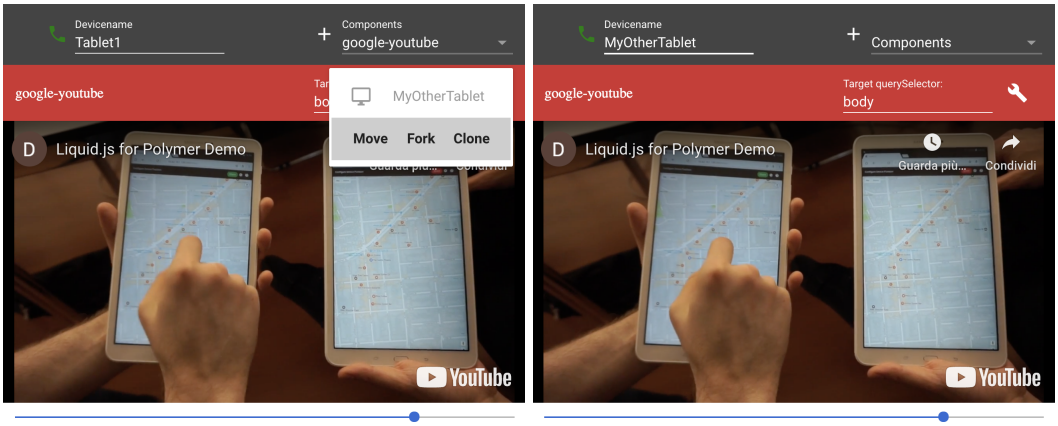
Listing 9.16. Definition of the `<liquid-component-slider>` component

```
1  <link rel="import" href="/paper-slider.html">
2  <dom-module id="liquid-component-slider">
3    <template>
4      <paper-slider value="{{currenttime}}"
5          max="{{duration}}"> </paper-slider>
6    </template>
7    <script>
8      Polymer({
9        is: 'liquid-component-slider',
10       behaviors: [LiquidBehavior],
11       properties: {
12         currenttime: { value: 0, liquid: true, notify: true },
13         duration: { liquid: true, notify: true }
14       }
15     });
16   </script>
17 </dom-module>
```

`currenttime` is propagated to the `<liquid-component-youtube>`, which takes care of updating its own liquid property and therefore triggers the `_timeChange` observer.

At line 15 we inject two behaviors into the component: the `LiquidBehavior` which gives access to the LUE primitives; and the `LiquidContainerBehavior`, which annotates a component as a container. The developers do not need to do anything more than injecting the behavior in order to make the component a liquid container. At line 22 we define the `ready` method, which Polymer automatically invokes when the component is instantiated and ready. Inside this function, as a precaution, we pre-load the component type of `<liquid-component-slider>`, in order to be sure that the component can be instantiated on the targets of the migrate, fork, and clone LUE primitives.

In Listing 9.16 we show the definition of the `<liquid-component-slider>`. At line 1 we import the `<paper-slider>` component [Goo17c], which we load at line 4 in order to implement the slider input of the `<liquid-component-slider>` itself. We bind the slider to two liquid properties: `currenttime`, which represents the number of seconds currently playing on the video (defined at line 12), and `duration`, which represents the maximum length of the video in seconds (defined at line 13). The value for both properties is passed from the parent of the slider component. At line 10 we can see that the component loads the

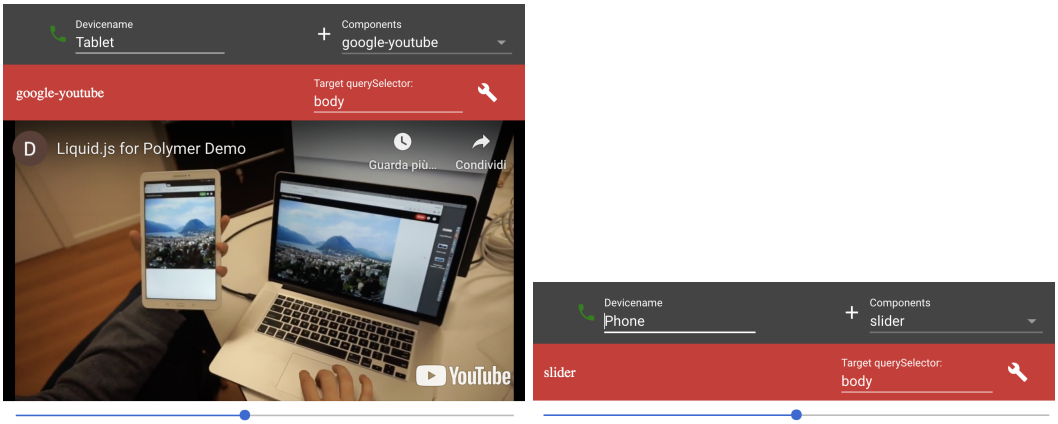


(a) Tablet 1 (b) Tablet 2

Figure 9.18. Multi-device deployment of `<liquid-component-youtube>` on two tablets

`LiquidBheavior`, in this case it is not necessary to inject any other behavior, since the parent will be able to detect without any additional annotation when this component is one of its children.

In Figure 9.18 we show the deployment of the `<liquid-component-youtube>` on two tablets. The application is initially deployed and instantiated on *tablet 1*. The black topbar on top of the device is an instance of the `liquid-create` component, which can be used for debugging purposes and allows to select loaded components from a drop-down menu and instantiate them in the *body* of the Web application. This component also allows the developers/users to set a *device name*, which can be helpful for recognizing remote devices. The red bar on top of the `<liquid-component-youtube>` is a liquid UI wrapper. The wrapper displays the name of the wrapped component (e.g., google-youtube in this example) and allows the users to display a drop-down menu containing the list of all other connected devices by clicking on the *wrench* button. The users can then select a LUE primitive and invoke it on target device. Optionally, the wrapper allows the users to pass as a string parameter the QuerySelector pointing to a HTML element contained on the target device (e.g., *body* in this example). In Figure 9.18a the users decide to clone the component targeting the second tablet as shown in Figure 9.18b. When the clone primitive is invoked, Liquid.js creates a new instance of the `<liquid-component-youtube>` on tablet 2, and, since the component imports the `LiquidContainerBehavior`, then it also create a new instance of the `<liquid-component-slider>` on target device. Liquid.js transparently takes



(a) Tablet (b) Phone - horizontal

Figure 9.19. Multi-device deployment of `<liquid-component-youtube>` on a tablet and a phone

care of pairing the liquid properties of both components accordingly on both tablets. The users can now stop and play the video from either devices and propagate the effect to the other tablet. Similarly they can change the value of the slider and propagate the new time to both video viewers.

In Figure 9.19 we show that it is still possible to individually clone the child slider to a remote device. In this case we clone the slider from tablet 1 (Figure 9.19a) to a smartphone (Figure 9.19b). Liquid.js instantiates a new slider component on the smartphone and pairs the liquid properties `currenttime` and `duration`.

9.4.4 Liquid UI Wrappers and Position-Aware Primitives

The liquid UI wrappers are optional tools meant to enhance the LUE. In the previous examples we already showed some wrappers that can help the users invoke and select a target for the LUE primitives. Liquid.js natively provides the `debug` and `default` liquid UI wrappers, however the developers can create any ad-hoc solution for their Web applications.

In Figure 9.20 we show the `debug` wrapper. This wrapper allows the developers (or the users) to access the migrate, fork, and clone primitives. Depending on the icon, the developers have access to multiple features: • the phone icon allows the developers to have a feedback on the current state of the connection with the signalling server. The icon can be green or red: whenever the icon is green, the Web application is currently connected to the Liquid.js signalling server, otherwise it is red; • the four-arrows icon allows the users to call the *clone* LUE primitive with a drag-and-drop gesture which we will demonstrate later in this section; • The label displays the name of the wrapped component (e.g., *text*); • The drop-down menu can be used to select a device in the list of all connected devices, once the device is selected the users can use the next to buttons for accessing the *fork* and *migrate* LUE primitives; • The copy button can be used to *fork* the component on the selected device; • The cut button can be used to *migrate* the component on the selected device; • the `X` icon can be used to delete the instance of the component. The `debug` wrapper also allows the users to target the LUE primitives to the current device they are using. Typically it is unlikely that the users migrate or clone a component on the local device, however since the local device is a valid target for the migrate, fork, and clone primitives, the wrapper provides this possibility. The `default` wrapper which we will show later hides the local device from the list of available devices.

In Figure 9.21 we show the `default` wrapper. Similarly to the `debug` one, the `default` wrapper is rendered as the black bar on top of the liquid component which it is wrapping (see Figure 9.21a). By clicking on the *wrench* icon the users have access to a cleaner drop-down menu where they can select both the target device and the LUE primitives. From the menu the users can also broad-

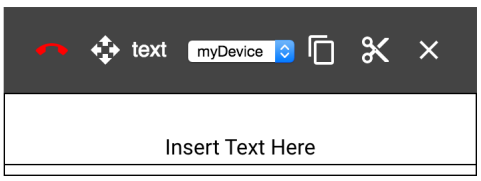
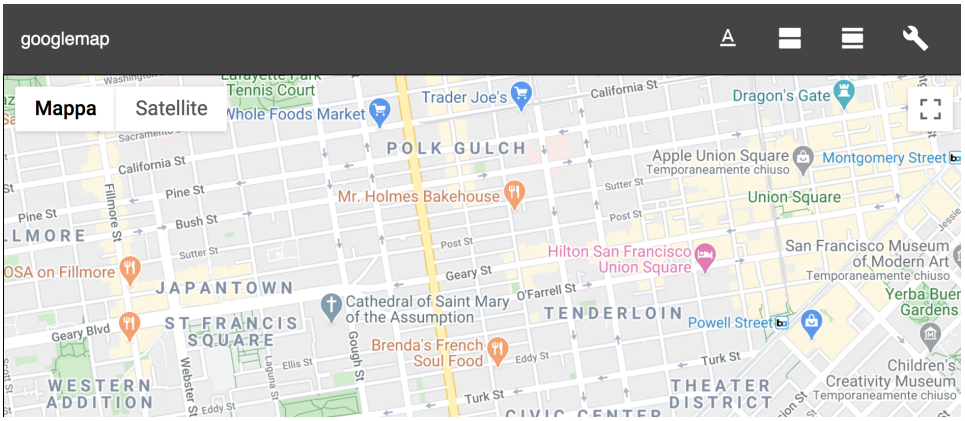
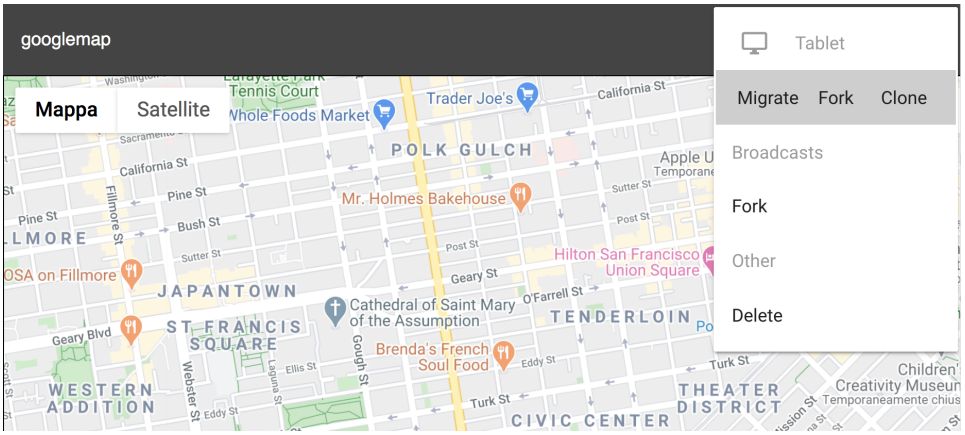


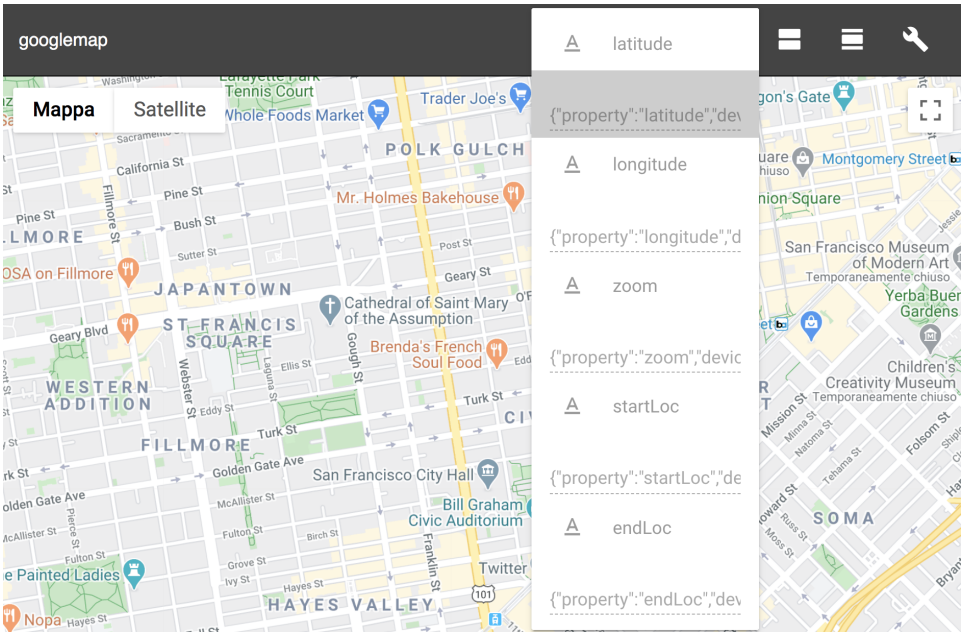
Figure 9.20. `Debug` liquid UI wrapper on top of `<liquid-component-text>`



(a) Wrapper on top of the component



(b) LUE primitives and target devices



(c) Liquid URIs

Figure 9.21. `Debug` liquid UI wrapper on top of `<liquid-component-googlemap>`

cast the forking and cloning of the component to all connected device (shown in Figure 9.21b). Without the broadcast feature, the users need to invoke the selected primitive for each device individually.

The three icons on the left of the wrench can be used to investigate the liquid state of the component. By clicking on any of the three icons, a drop-down menu appears displaying the summary of all liquid properties defined inside the liquid component. The users can click on the cell below the name of the liquid property for copying on the clipboard its liquid URI. Depending on the icon clicked the users can access the list of properties grouped by type: 1. The first icon groups all Boolean, Number, and String properties; 2. The second icon displays Array properties; 3. The last icon displays Object properties.

Liquid UI wrappers do not need to look similar to the `debug` and `default` wrappers, and developers can design and implement their own ad-hoc solutions. We now show a more complex example where we build a liquid UI wrapper that allows the users to target the devices depending on their position, instead of using a drop-down menu [156]. In Figure 9.22 we show a Web application that allows the users to create multiple chat rooms on the same page. For every chat

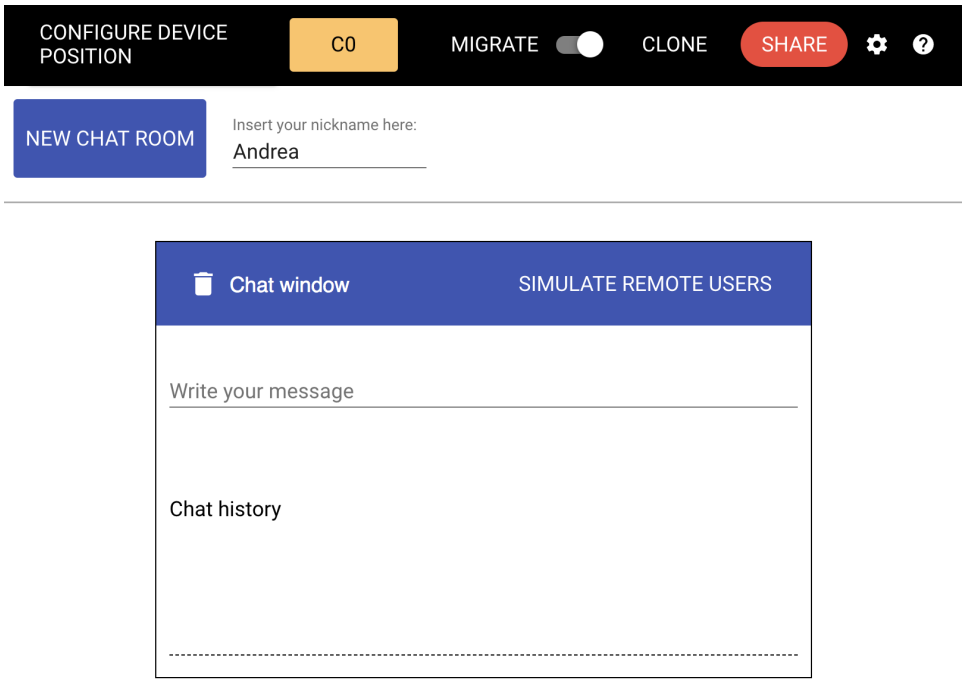
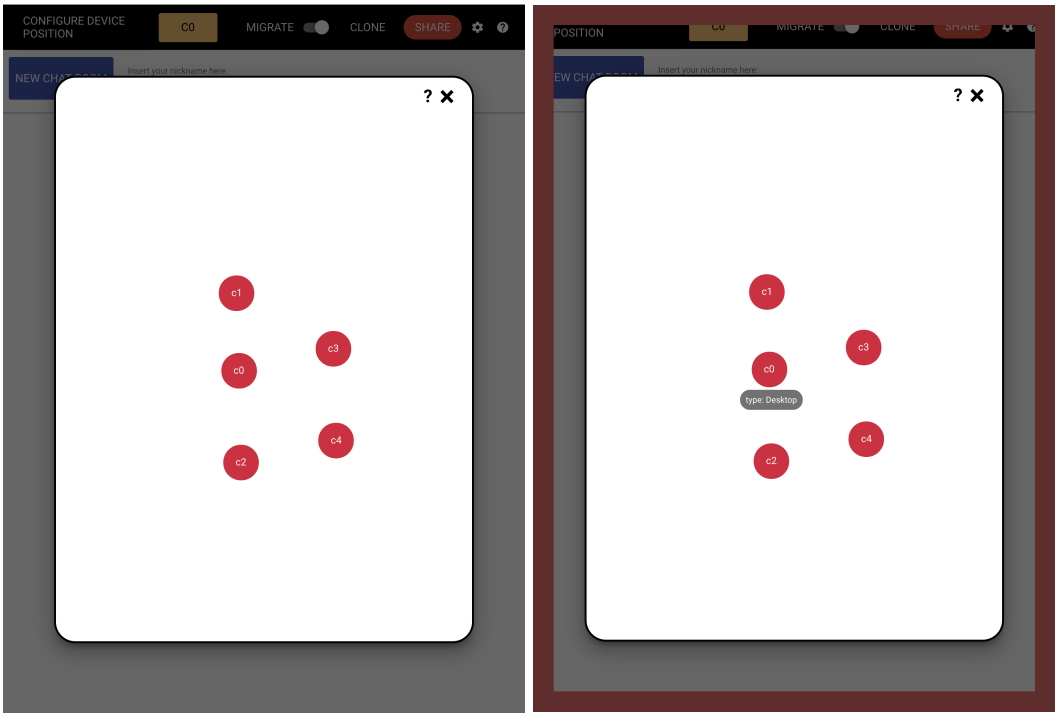


Figure 9.22. `<liquid-space>` and `liquid-component-chat` components



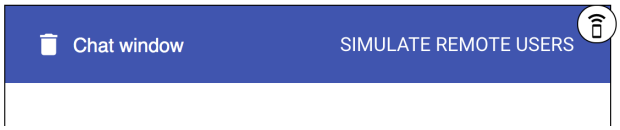
(a) Map view (b) Hovering on a device

Figure 9.23. `<liquid-space>` position configuration

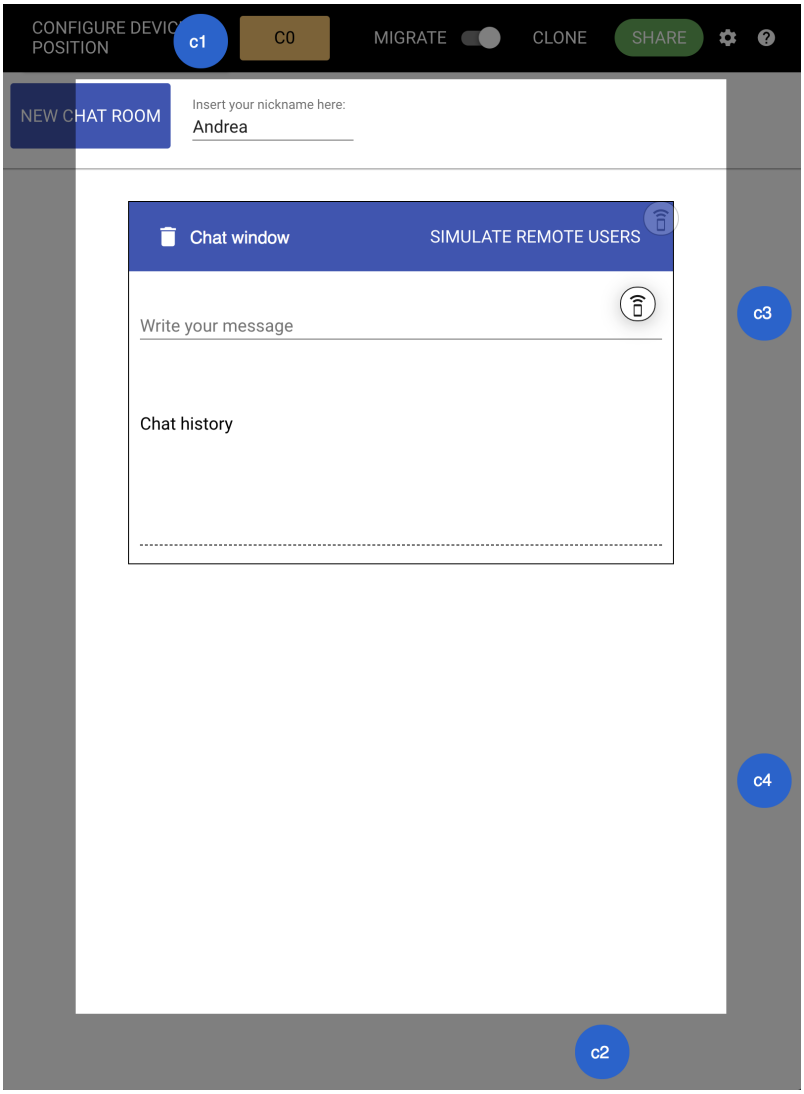
room, Liquid.js instantiates a new instance of the `liquid-component-chat` in the body of the Web application. The wrapper is displayed in blue on top of the chat component. In the current state of the application shown in the example, the wrapper provides to the users the name of the component, the *bin* icon that allows to delete the component instance by invoking the Liquid.js API, and the *simulate remote users* feature which is part of the chat room implementation. The top black bar of the Web application is an instance of the `<liquid-space>` component, which is bound to all liquid UI wrappers loaded in the page.

The `<liquid-space>` component can be used to configure the relative positions of the connected devices by clicking on the button labeled *configure device position*. In Figure 9.23 we show what happens when the users click on this button. Upon clicking a modal window containing an empty white space with multiple colored circles appears in front of the Web application (see Figure 9.23a), each circle represents a connected device and it displays the device name in the center. In this example there are five connected devices named `c0`, `c1`, `c2`, `c3`, `c4`. The device names are displayed and can be edited on each device by clicking on the yellow rectangle shown in the `<liquid-space>` component. The users

(a) Users can click on the share button.



(b) When the share button is clicked, a new icon appears on the top right corner of all chat room components.



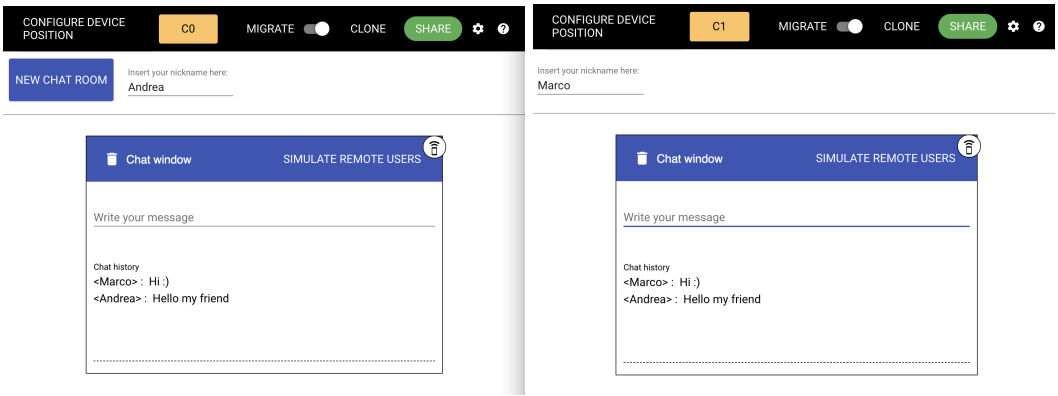
(c) When the users drag the icon, a gray frame appears around the Web application containing multiple circle icons representing the connected devices. If the users configured the relative positions of the devices, the circles are displayed in the frame in the correct relative direction from the current device viewpoint. The users can drop the icon on the desired device circle in order to perform the LUE primitive selected.

Figure 9.24. `<liquid-space>` and liquid UI wrapper

can drag and drop the circles in any empty position of the white space of the modal window, replicating the relative position of the devices in the surrounding space. E.g., `c1` is in front of `c0` and `c3` is positioned on the right of `c0`. In order to simplify as much as possible the identification of the devices on the map, the users can hover the mouse on any of the circles like shown in Figure 9.23b, in turn the `<liquid-space>` component will show the corresponding type of the device below it, and display an orange frame around the borders of the hovered device page.

Once the users finish positioning the devices on the map, they can finally close the modal window and start migrating or cloning the instances of `liquid-component-chat` among the connected devices. In Figure 9.24 we show how the users can now use the relative positions of the devices in order to select the target of the migrate and clone LUE primitives. When the users click on the share button (Figure 9.24a) it becomes green and an hidden icon appears on the top right corner of all chat instances (Figure 9.24b). The icon is draggable and whenever the users start the dragging operation a dark gray frame appears around the Web page (Figure 9.24c). Similarly to the modal window map, this frame contains multiple colored circles representing the connected devices. The position of the circles depends on the relative position configured in the map, and they are placed in the correct relative direction from the active device perspective. The users can then drag and drop the icon on the circle representing the desired target device of the LUE primitive, which will transparently query Liquid.js to perform the required action.

In Figure 9.25 we show the final deployment of the chat application, after the user on `c0` (Figure 9.25a) dragged and dropped the UI wrapper icon on top



(a) Laptop 1 (b) Laptop 2

Figure 9.25. `<liquid-component-chat>` after cloning

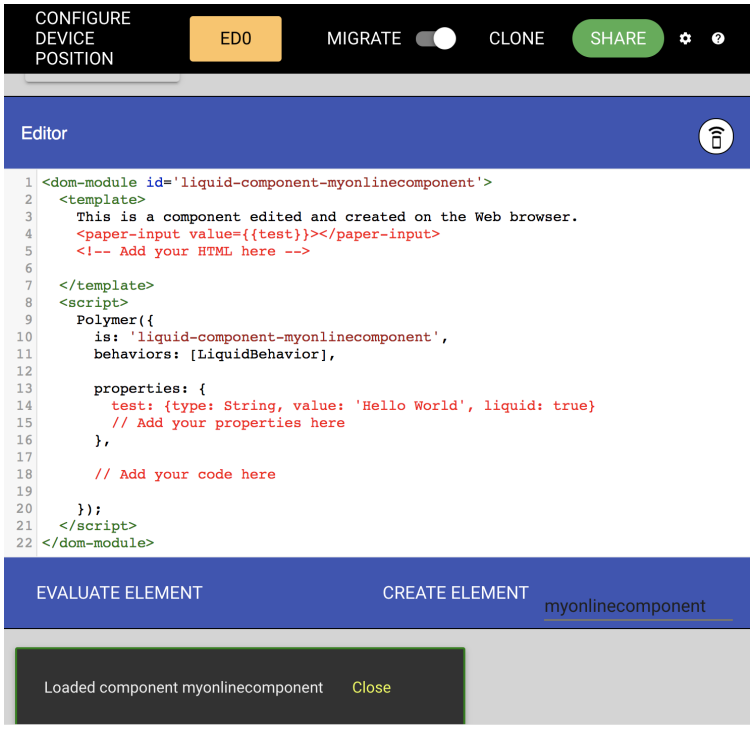
of the icon representing device `c1` (Figure 9.25b). The component is cloned on the target device and the two users can now chat.

9.4.5 Experiment: Creating Liquid Components on the Clients

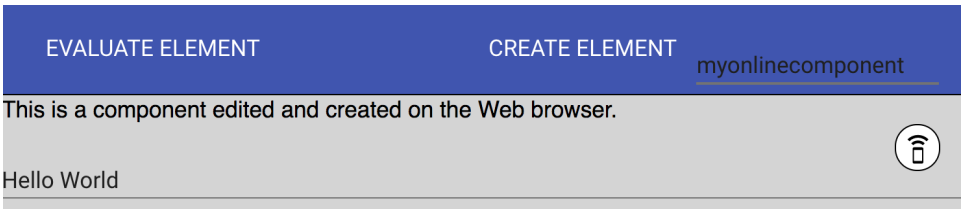
In this section we present an experimental example that demonstrates the power of the decentrilized approach of Liquid.js. This Web application can be used to design new liquid components on the client-side that can be moved among clients without uploading them on a Web server. The Web application is meant to be an experiment to showcase the features of Liquid.js and should not be used in a production environment without improving its security. We understand that its current implementation can be used by malicous users in a multi-user scenario. The application is currently meant to be used in a single-user scenario, or in a multi-users scenarios if all users trust each other.

The Web application is composed by a single liquid component called `editor`. In Figure 9.26 we show how `<liquid-component-editor>` looks when it is instantiated on a desktop computer togheter with its liquid UI wrapper and the

Figure 9.26. `<liquid-component-editor>`



(a) Evaluate



(b) Instantiate

Figure 9.27. Create an instance of the component on the client-side

`<liquid-space>` component shown on top. The editor is composed by three main elements: • the writable area in which the users can design their own liquid components; • the *evaluate element* button, which loads and evaluates the component source into the Web application (this operations uses the `eval` function available in the Web browser); • the *create element* button connected to a text-area which allows to instantiate any pre-loaded liquid component by providing its name.

The users can design any liquid component inside the editor. In this example the component we designed in the text-area is similar to the `<liquid-component-example>` we built in Section 9.4, but it also add a unique

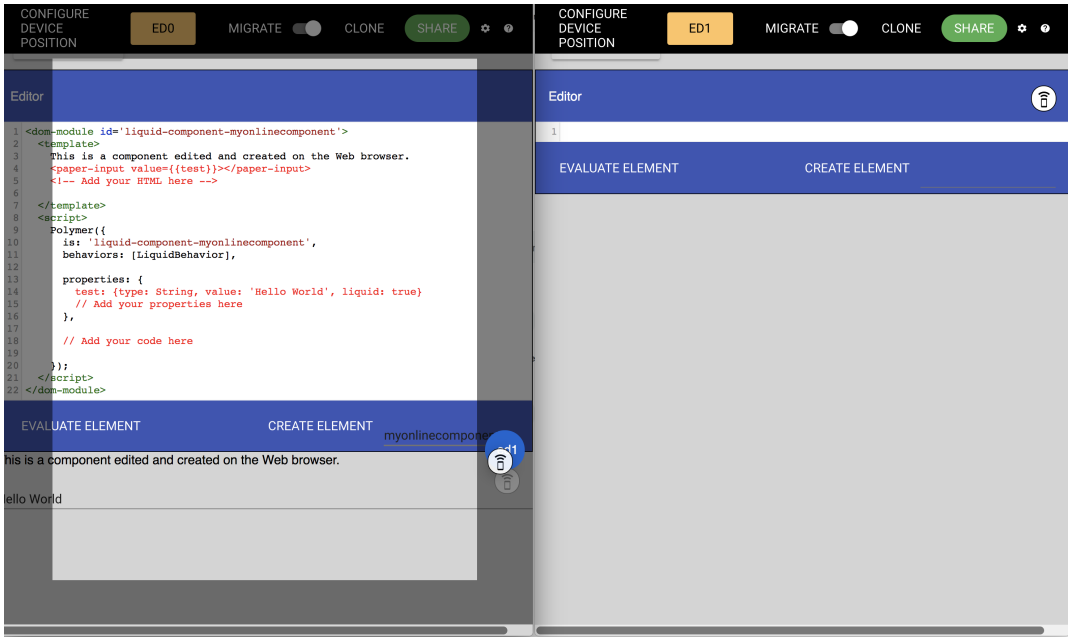


Figure 9.28. Drag and drop cloning of `my-online-component` from device 1 (on the left) to device 2 (on the right).

label. In Figure 9.27 we show how the users can instantiate the component in the Web browser. When the code they write is correct and it defines a valid Polymer component, they can press the *evaluate* button (Figure 9.27a). If the evaluation is successful, a green-bordered toast message appears on the bottom left of the page prompting the users that the were no issues with the evaluation, otherwise a red-bordered message appears and they need to correct the issues in the editor. When the editor evaluates the new component, Liquid.js stores its definition in the local client-side database where it also stores all the liquid components it previously downloaded (either from the Web server or from other clients). This means that Liquid.js considers the new component a full-fledged liquid component and it in not important if it was created on the Web browser or downloaded from somewhere else. Every time the users try to instantiate a component with the specified name, they will append the component on the body of the Web application. In Figure 9.27b the users can press the *create* button and create an instance of the new component underneath the editor. In this particular example, the instantiated component in boundle with the drag-and-drop migration UI wrapper.

If the users connect an additional device to the Web application, they can now move the component by dragging-and-dropping the component into the

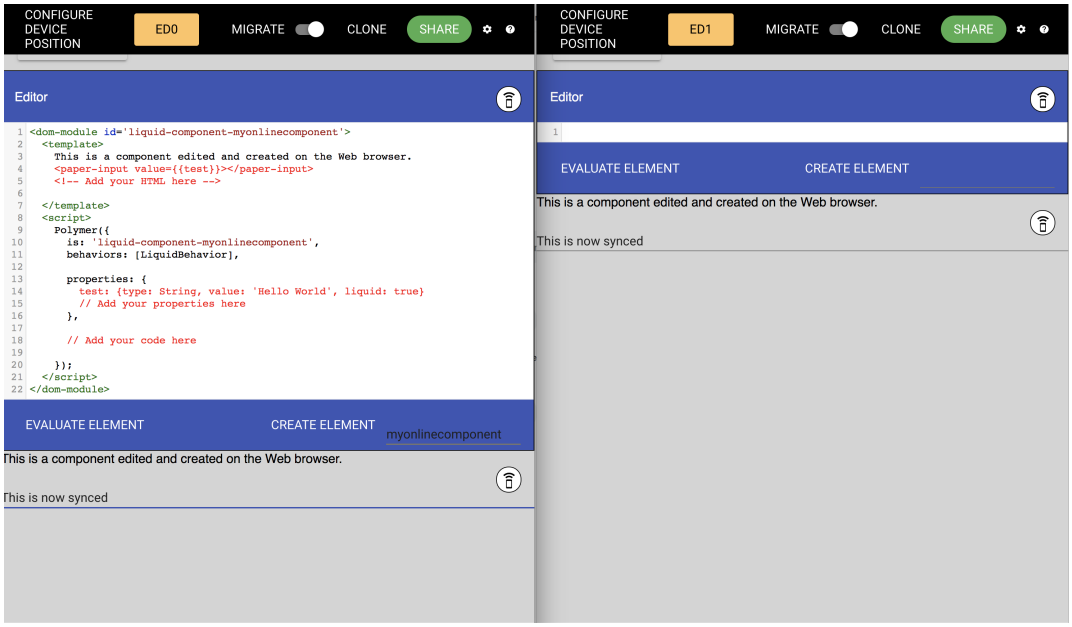


Figure 9.29. `my-online-component` cloned and paired among two devices

neighbour device (Figure 9.28). In our example, the users clone the component from the device on the left, to the device on the right, which initially does not hold the source of `my-online-component`. When the clone operation is performed, the framework notices that the device on the right does not own the component source, therefore it also sends a copy of the component definition to the target device.

In Figure 9.29 we can see the final deployment of the application when the clone process ends. The component is now loaded and instantiated on the second device and the liquid properties are synchronized.

The interesting part of the experiment is that the users can now disconnect from the Web server of the application without reloading the page (e.g., they can close the Node.js process running the server) and they can repeat all the steps discussed above and recreate the same behavior. Disconnecting from the Web server without reloading the page, allows the Web browsers to maintain the WebRTC channel connecting the clients even if they lost connection with the signalling server. When the users create a new component and move it from a device to another, they can be sure that the component is never uploaded to the Web server.

Chapter 10

Conclusion

10.1 Summary

In the past three decades the Web underwent a fast evolution and now became a mature platform that allows the development of powerful Web applications. The latest HTML5 standards and the new proposed Web technologies are shifting the center of the Web towards the edge, making it possible to run distributed Web applications in any standard-compliant Web browser. As the users own a constantly increasing number of devices and the Web applications are growing more and more complex, the users' expectation is to be able to access them from any of their devices at any given time without the effort of manually managing them. Now, it is more important than ever that developers understand how the set of connected devices can support each other in a distributed mobile Web environment concurrently running a Web application.

In this dissertation we presented the design of decentrilized cross-device liquid Web applications and the Liquid.js for Polymer framework. In the current state of the art of liquid software, many of the proposed solutions add the Liquid User Experience to existing Web applications by extending the view layer with cross-device interface features. These approaches usually update the front-end without touching or re-designing the back-end or the data and logic layers of the Web application. While from the users perspective they may perceive a liquid-like user experience, the software itself does not implement all the features expected by the liquid software manifesto. We believe that in order to create liquid applications able to take full advantage of the set of connected devices running on multiple Web browsers, we must consider liquid software requirements from the beginning and re-design all the layers of a Web application (data, logic, and view).

The contributions presented in this dissertation can be summarized as follows:

- **RQ#1** Liquid Software Design - "How can we help Web developers design liquid software and the Liquid User Experience?"

In order to answer this research question we first investigated the evolution of liquid software in Chapter 2, then presented and explained the liquid software requirements in Section 2.1. We investigated the most famous solutions existing in the industry in Section 2.4 and the experiments, prototypes, and studies proposed in multiple research areas.
The main design contributions for liquid Web applications are presented in Chapter 3, where we defined the design space of liquid software (Section 3.2). The design space is based on twelve architectural dimensions (Section 3.1) related to liquid software: topology, application source topology, state replication topology, layering, client deployment, granularity, state identification, synchronization, device usage, UI adaptation, primitives, and discovery. Developers can use the proposed alternatives to understand the complexity of liquid applications and plan how to build their own. In the section we also discussed real-world applications and solutions featuring the same design considerations.
In the maturity model presented in Section 3.3 we investigated the quality attributes related to liquid software built on the Web: latency, LUE primitives, UI adaptation, and privacy. These quality attributes are also related to non-Web liquid software and can be considered in the design of liquid software which is built on top of other platforms (e.g., native clients that are not specifically built on a HTML5-compliant Web browser).
In the remainder of this dissertation, specifically in Chapters 4, 5, and 6, we investigated the liquid software design by discussing multiple technological alternatives available on the Web. For each technology we presented their trade-offs when multiple choices were available.

- **RQ#2** Beyond Centralized Deployments - "How can we abstract liquid Web applications away from the current centralized deployment approaches?"

In this dissertation we presented the rationale behind the decision of using the Web platform for creating liquid software able to overcome the boundaries of vendor-locked OSs and hardware. The choice of the Web platform follows the evolution of the available HTML5 standards, which established a platform that can be used to create decentralized and distributed Web applications.
In Section 3.1 we investigated the design space of liquid software and explained which architectural concerns are related and should be considered in the development of liquid applications. These architectural concerns also apply on Web-based architectures. In the maturity model in Section 3.3 we focused our study

on the Web platform and presented the reasons why liquid software evolved in the past decades on the Web. We designed the maturity model for understanding how liquid applications can provide support for the Liquid User Experience in multiple deployment configurations and developed with different Web technologies. Each level of the maturity model is related with quality attributes that can be implemented only if the architectural decisions of the liquid Web applications meet the maturity level requirements. Furthermore we discussed all the Web technologies and standards that can be used for reaching higher level of maturity. Developers can use the presented design insights for developing their own liquid Web software.

- **RQ#3** Liquid User Experience Adaptation Among Devices - "How can we make the Distributed User Interfaces of a Web application automatically adapt to the set of connected devices?"

The automatic adaptation of the UI of a liquid application for enhancing the LUE is mainly presented in Chapter 6, where we proposed an extension for the standard CSS3 media queries. We designed ad-hoc CSS3 media types, features, and the syntax for allowing developers to create rules that can be used to automatically adapt the view layer of a cross-device liquid Web application. We also designed the adaptation algorithms and events that are needed to be considered for implementing the automatic computation of new deployments dynamically at runtime.

The Liquid.js for Polymer framework we implemented and presented in Chapter 7 is meant to demonstrate how our design of liquid media queries can be implemented in a Web framework. Specifically in Section 7.5 we built the liquid media queries inside the behavior of a Polymer component and simulated the rules we designed with the `<liquid-style>` component. We also presented how it is possible to decentralize the adaptation algorithm in Liquid.js and shift the whole computation on the client-side, without the need of relaying messages through a Web server. Finally in Chapter 8 we also showed the exposed API methods developers can use in order to create their own Liquid User Experience (LUE), and in Section 9.3 we show the expressiveness of the liquid media queries with two examples.

- **RQ#4** Resource Sharing Among Devices - "How can we take advantage of all resources provided by the set of connected devices?"

In order to answer this RQ we intentionally separated the description of the *design* chapters into three independent layers: data, logic, and view. The layers can be mapped to specific resources provided by each connected device: storage, CPUs, and screens. While the view layer and the provided screen resource are covered by RQ#3, in this RQ we focused on the provided storage and CPUs

resources.
In Chapter 4 we designed the liquid data layer, which can be used to transparently synchronize data and state across the set of connected devices. We presented the technologies and framework that can be used for creating a decentralized or distributed synchronization mechanism. We also presented data privacy and model rules for enhancing the protection of data created by the users. The liquid data layer is at the core of the Liquid.js implementation (Chapter 7) and is evaluated in Section 9.1. We built and evaluated two different network strategies that can be used in data synchronization and inter-device communications in a liquid application with devices that have access to limited bandwidth or storage resources.
In Chapter 5 we designed Liquid WebWorkers (LWWs) for creating a liquid logic layer and modelled the horizontal computation offloading of stateless tasks across multiple devices. We also demonstrated how a microbenchmark can be used to evaluate the resources available in a Web browser without having direct access to the hardware specifications. In Section 7.4 we presented the implementation of the LWWs built inside the Liquid.js framework and evaluated them in Section 9.2, together with the evaluation of the micro-benchmark.

- **RQ#5** Privacy and Security - "How can we design secure liquid Web applications? How can we enhance privacy?"

Multiple times throughout this dissertation we claimed that the users must be in control of their data and should be able to decide if a particular device should or should not be used in the cross-device deployment.
In Section 7.6 we modelled a Discretionary Access Control (DAC) system for enhancing privacy and security in a liquid Web application. We presented the attacks and possible failures in an unprotected application and presented a decentralized solution for protecting the users' data. While the model is meant to run on the client-side of the application, the identity of the users are stored in a central Web server.

10.2 Future Work

The main goal of this dissertation was to provide to Web developers the necessary knowledge required for developing their own liquid Web applications. Our contributions and artefacts can be used to understand the complexity of the development of liquid Web software and allows Web developers to make better design decisions thanks to our discussion on multiple design and technological alternatives.

So far Liquid.js for Polymer was used by students enrolled in the second year

bachelor at Università della Svizzera italiana (USI) for creating their own liquid applications during the project of the *Web Atelier* course. While they were able to successfully build liquid Web applications with the framework, we believe that in the future we need to study how developers with different levels of experience build their own solutions. In our particular scenario, the students found it difficult to shift from a server-centric approach to a decentralized one, even if the proposed Liquid.js API helped them create their own LUE. We believe that more experienced developers may have different difficulties and different needs and that by studying liquid applications together with multiple Web developers we can better understand their needs and create debugging tools suited for creating liquid software.

The Web is still evolving and as this dissertation is being written new HTML5 standards are constantly proposed [Wil20]. We believe that the next generation of liquid software should be implemented in the Web and that they will shift away from vendor-locked centralized solutions towards decentralized options that do not necessarily rely on Cloud services. Liquid Web applications can be further studied towards many directions, in particular we should focus on version control and distribution:

- As liquid Web applications can exist in the Web browsers without the need of querying the Web server after the initial download, we need to address the versioning of liquid components. During the lifespan of the Web application on a specific device, a new version can be released and downloaded by other devices. When those devices interact with each other, they can potentially incur into issues or glitches. Liquid.js, and liquid Web applications in general, should be able to detect when there is a mismatch in the application versions and automatically download the newest one, either from the Web server of from the paired Web browser. Version control adds new challenges in the design of liquid Web applications, such as: – Can the users be able to use an older version of the application if they want? – Can the users download multiple versions of the same application and use them sequentially? – How can the Web browser store and discard different versions of the Web application?

- While we already experimented with Internet of Things (IoT) devices in our related work [52], our solution for including embedded devices and sensors in the set of connected devices of a liquid application is strongly centralized. Fischer presents in his thesis an adapter module that can be locally loaded or remotely connect to an instance of a signalling server of Liquid.js. The module allows any kind of micro-processor or sensor to connect to the liquid application as if they were a Web browser instantiating a single liquid component defining any number of liquid properties. Those simulated Web browsers however do not

have access to the specification of the WebRTC DataChannels as a standard Web browsers does, and therefore all messages exchanged between the IoT devices and the connected Web browsers are relayed thorugh the signalling server (using WebSockets). While this approach at the time was the only possible solution for making Web browsers and IoT devices communicate, nowadays the HTML5 Web Bluetooth [Web17] standard improved and can be used to connect Bluetooth-enabled devices directly with a Web application. We envision that in the future, thanks to Web Bluetooth or other similar technologies, it will be possible to create liquid Web applications that do not require a signalling server. The same principle can be extended from IoT devices to public displays.

- Similarly, the Discretionary Access Control model we proposed for enhancing privacy and security, uses a central server for storing the identity of the users connecting to the Web application. We believe that this approach can be further decentralized and fully distributed on the client-side of the Web application by storing the identity of the users in a block-chain. We should investigate if the block-chain approach has an impact on the performance of the Liquid User Experience in comparison to the server-centric approach.

As a concluding remark, in this dissertation we presented Liquid.js, a liquid sfotware framework that can be used to build liquid applications on the Web, however, thanks to our personal experiences, we believe that we need to gather more insights from both developers and users before it can be used at its full potential in a real-world scenario. In fact we must teach the developers how to design and implement decentralized applications that shift away from the traditional server-centric architectures. Moreover from the users perspective, it is not always clear how to implement an intuitive Liquid User Experience that can be used by any user. New kind of interaction and gestures should be studied in order to enhance the migration process of liquid Web applications.

Bibliography

[1] *OLSR*. `https://www.ietf.org/rfc/rfc3626.txt` (cit. on p. 145).

[2] Edward Anstead, Steve Benford, and Robert J Houghton. “Many-Screen Viewing: Evaluating an Olympics Companion Application”. In: *Proceedings of the International Conference on Interactive Experiences for TV and Online Video*. ACM. 2014, pp. 103–110 (cit. on p. 106).

[3] David P Anderson and Gilles Fedak. “The Computational and Storage Potential of Volunteer Computing”. In: *Proceedings of the International Symposium on Cluster Computing and the Grid*. Vol. 1. IEEE. 2006, pp. 73–80 (cit. on p. 28).

[4] Martin L Abbott and Michael T Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Pearson Education, 2009 (cit. on p. 17).

[5] Ejaz Ahmed et al. “Process State Synchronization for Mobility Support in Mobile Cloud Computing”. In: *Proceedings of the International Conference on Communications (ICC)*. IEEE. 2017, pp. 1–6 (cit. on p. 4).

[6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. “The Internet of Things: A Survey”. In: *Computer Networks* 54.15 (2010), pp. 2787–2805 (cit. on pp. 25, 106).

[7] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the Definitive Guide: Time to Relax*. " O’Reilly Media, Inc.", 2010 (cit. on p. 56).

[8] Florian Alt et al. “Designing Shared Public Display Networks–Implications From Today’s Paper-Based Notice Areas”. In: *Proceedings of the International Conference on Pervasive Computing*. Springer. 2011, pp. 258–275 (cit. on p. 26).

[9] Matti Anttonen et al. “Transforming the Web into a Real Application Platform: New Technologies, Emerging Trends and Missing Pieces”. In: *Proceedings of the ACM Symposium on Applied Computing*. 2011, pp. 800–807 (cit. on p. 16).

[10] Muhammad Raisul Alam, Mamun Bin Ibne Reaz, and Mohd Alauddin Mohd Ali. “A Review of Smart Homes — Past, Present, and Future”. In: *IEEE transactions on systems, man, and cybernetics, part C* 42.6 (2012), pp. 1190–1203 (cit. on p. 15).

[11] Mario Barbacci et al. *Quality Attributes*. Tech. rep. Software Egineering Institute Carnegie Mellon University, 1995 (cit. on p. 54).

[12] Victoria Bellotti and Sara Bly. “Walking Away from the Desktop Computer: Distributed Collaboration and Mobility in a Product Design Team”. In: *Proceedings of the Conference on Computer Supported Cooperative Work*. ACM. 1996, pp. 209–218 (cit. on pp. 4, 24).

[13] Sriram Karthik Badam and Niklas Elmqvist. “Polychrome: A Cross-Device Framework for Collaborative Web Visualization”. In: *Proceedings of the International Conference on Interactive Tabletops and Surfaces*. ACM. 2014, pp. 109–118 (cit. on pp. 31, 68, 69).

[14] Federico Bellucci et al. “Engineering Javascript State Persistence of Web Applications Migrating Across Multiple Devices”. In: *Proceedings of the SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM. 2011, pp. 105–110 (cit. on pp. 67, 69).

[15] Eric A Benson. *Use of Browser Cookies to Store Structured Data*. US Patent 6,714,926. 2004 (cit. on p. 56).

[16] Tim Berners-Lee. *Re-Decentralizing the Web - Some Strategic Questions*. Keynote Address at Decentralized Web Summit. 2016 (cit. on pp. 21, 171).

[17] Tim Berners-Lee, Mark Fischetti, and Michael L Foreword By-Dertouzos. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*. HarperInformation, 2000 (cit. on p. 64).

[18] Masiar Babazadeh, Andrea Gallidabino, and Cesare Pautasso. “Liquid Stream Processing Across Web Browsers and Web Servers”. In: *Proceedings of the International Conference on Web Engineering (ICWE2015)*. Springer. 2015, pp. 24–33 (cit. on p. 20).

[19] Nilton Bila et al. “Pagetailor: Reusable End-User Customization for the Mobile Web”. In: *Proceedings of the International Conference on Mobile Systems, Applications and Services*. ACM. 2007, pp. 16–29 (cit. on pp. 66, 69).

[20] Peter Brusilovsky and Mark T Maybury. “From Adaptive Hypermedia to the Adaptive Web”. In: *Communications of the ACM* 45.5 (2002), pp. 30–35 (cit. on p. 17).

[21] Flavio Bonomi et al. “Fog Computing and its Role in the Internet of Things”. In: *Proceedings of the Workshop on Mobile Cloud Computing*. ACM. 2012, pp. 13–16 (cit. on pp. 25, 28).

[22] Daniela Bourges-Waldegg et al. “The Fluid Computing Middleware: Bringing Application Fluidity to the Mobile Internet”. In: *Proceedings of the Symposium on Applications and the Internet*. IEEE. 2005, pp. 54–63 (cit. on pp. 20, 39).

[23] Daniele Bonetta and Cesare Pautasso. “An Architectural Style for Liquid Web Services”. In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA2011)*. IEEE. 2011, pp. 232–241 (cit. on p. 20).

[24] Harry Brignull and Yvonne Rogers. “Enticing People to Interact with Large Public Displays in Public Spaces”. In: *Proceedings of INTERACT*. Vol. 3. Brighton, UK. 2003, pp. 17–24 (cit. on p. 15).

[25] A Bouzid and D Rennyson. *The Art of SaaS: A Primer on the Fundamentals of Building and Running a Successful SaaS Business*. 2015 (cit. on p. 30).

[26] Robert Bradford et al. “Live Wide-Area Migration of Virtual Machines Including Local Persistent State”. In: *Proceedings of the International Conference on Virtual Execution Environments*. ACM. 2007, pp. 169–179 (cit. on p. 53).

[27] Alexandru Boicea, Florin Radulescu, and Laura Ioana Agapin. “MongoDB vs Oracle – Database Comparison”. In: *Proceedings of the International Conference on Emerging Intelligent Data and Web Technologies*. IEEE. 2012, pp. 330–335 (cit. on p. 56).

[28] Eric Brewer. “CAP Twelve Years lLter: How the "Rules" Have Changed”. In: *Computer* 45.2 (2012), pp. 23–29 (cit. on p. 52).

[29] Frederik Brudy et al. “Cross-Device Taxonomy: Survey, Opportunities and Challenges of Interactions Spanning Across Multiple Devices”. In: *Proceedings of the Conference on Human Factors in Computing Systems*. ACM. 2019, pp. 1–28 (cit. on p. 22).

[30] Gianluca Brugnoli. “Connecting the Dots of User Experience”. In: *Journal of Information Architecture* 1.1 (2009) (cit. on p. 4).

[31] Josiah L Carlson. *Redis in Action*. Manning Publications Co., 2013 (cit. on p. 56).

[32] Diane J. Cook and Sajal K. Das. “How Smart are our Environments? An Updated Look at the State of the Art”. In: *Pervasive and Mobile Computing* 3.2 (2007), pp. 53–73 (cit. on p. 25).

[33] Sven Casteleyn, Irene Garrig’os, and Jose-Norberto Maz’on. “Ten Years of Rich Internet Applications: A Systematic Mapping Study, and Beyond”. In: *ACM Transactions on the Web (TWEB)* 8.3 (2014) (cit. on pp. 27, 44, 60, 65).

[34] Olexiy Chudnovskyy et al. “Awareness and Control for Inter-Widget Communication: Challenges and Solutions”. In: *Web Engineering*. Springer, 2013, pp. 114–122 (cit. on p. 26).

[35] Olexiy Chudnovskyy et al. “Inter-widget Communication by Demonstration in User Interface Mashups”. In: *Web Engineering*. Springer, 2013, pp. 502–505 (cit. on p. 26).

[36] Sarah Clinch et al. “Ownership and Trust in Cyber-Foraged Displays”. In: *Proceedings of the International Symposium on Pervasive Displays (PerDis2014)*. ACM. 2014, pp. 168–173 (cit. on p. 26).

[37] Sarah Clinch. “Smartphones and Pervasive Public Displays”. In: *IEEE Pervasive Computing* 12.1 (2013), pp. 92–95 (cit. on p. 25).

[38] Cinzia Cappiello, Maristella Matera, and Matteo Picozzi. “A UI-Centric Approach for the End-User Development of Multidevice Mashups”. In: *ACM Transactions on the Web (TWEB)* 9.3 (2015), pp. 1–40 (cit. on p. 26).

[39] M Scott Corson et al. “Toward Proximity-Aware InterNetworking”. In: *IEEE Wireless Communications* 17.6 (2010) (cit. on p. 26).

[40] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. “Designing Distributed Applications with Mobile Code Paradigms”. In: *Proceedings of the International Conference on Software Engineering (ICSE1997)*. Vol. 97. 1997, pp. 22–32 (cit. on pp. 17, 135).

[41] Brendan Cully et al. “Remus: High Availability Via Asynchronous Virtual Machine Replication”. In: *Proceedings of the Symposium on Networked Systems Design and Implementation*. San Francisco. 2008, pp. 161–174 (cit. on p. 53).

[42] Reginald Cushing et al. “Distributed Computing on an Ensemble of Browsers”. In: *IEEE Internet Computing* 17.5 (2013), pp. 54–61 (cit. on p. 28).

[43] Linda Di Geronimo, Maria Husmann, and Moira C Norrie. “Surveying Personal Device Ecosystems with Cross-Device Applications in Mind”. In: *Proceedings of the International Symposium on Pervasive Displays (PerDis2016)*. ACM. 2016, pp. 220–227 (cit. on p. 3).

[44] Linda Di Geronimo et al. “Ctat: Tilt-and-Tap Across Devices”. In: *Proceedings of the International Conference on Web Engineering (ICWE16)*. Springer. 2016, pp. 96–113 (cit. on p. 23).

[45] Hoang T Dinh et al. “A Survey of Mobile Cloud Computing: Architecture, Applications, and Approaches”. In: *Wireless Communications and Mobile Computing* 13.18 (2013), pp. 1587–1611 (cit. on pp. 25, 28, 44).

[46] Florian Daniel and Maristella Matera. “Turning Web Applications into Mashup Components: Issues, Models, and Solutions”. In: *Proceedings of the International Conference on Web Engineering*. Springer. 2009, pp. 45–60 (cit. on p. 26).

[47] Florian Daniel and Maristella Matera. *Mashups: Concepts, Models and Architectures*. Springer, 2014 (cit. on p. 26).

[48] David Dearman and Jeffery S Pierce. “It’s on my other Computer!: Computing with Multiple Devices”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI08)*. ACM. 2008, pp. 767–776 (cit. on p. 3).

[49] Niklas Elmqvist. “Distributed user interfaces: State of the art”. In: *Distributed User Interfaces*. Springer, 2011, pp. 1–12 (cit. on pp. 16, 24).

[50] Alan W Esenther. “Instant Co-Browsing: Lightweight Real-Time Collaborative Web Browsing”. In: *Proceedings of the International World Wide Web Conference (WWW2002)*. ACM. 2002 (cit. on p. 24).

[51] George Fairbanks. *Just Enough Software Architecture: A Risk-Driven Approach*. Marshall & Brainerd, 2010 (cit. on p. 54).

[52] Alexander Fischer. *Liquid Web of Things*. Sept. 2018 (cit. on pp. 142, 255).

[53] Ian Fette and Alexey Melnikov. *The WebSocket Protocol*. 2011 (cit. on p. 176).

[54] Luca Frosini, Marco Manca, and Fabio Paternò. "A Framework for the Development of Distributed Interactive Applications". In: *Proceedings of the SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM. 2013, pp. 249–254 (cit. on pp. 67, 69).

[55] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. "Understanding code mobility". In: *IEEE Transactions on Software Engineering* 24.5 (1998), pp. 342–361 (cit. on pp. 17, 34, 50).

[56] Ben Frain. *Responsive Web Design with HTML5 and CSS3*. Packt Publishing Ltd, 2012 (cit. on pp. 105, 106).

[57] Steven Feiner and Ari Shamash. "Hybrid User Interfaces: Breeding Virtually Bigger Interfaces for Physically Smaller Computers". In: *Proceedings of the Symposium on User Interface Software and Technology*. ACM. 1991, pp. 9–17 (cit. on p. 6).

[58] Alois Ferscha and Simon Vogl. "Pervasive Web Access Via Public Communication Walls". In: *International Conference on Pervasive Computing*. Springer. 2002, pp. 84–97 (cit. on p. 15).

[59] George H. Forman and John Zahorjan. "The Challenges of Mobile Computing". In: *Computer* 27.4 (1994), pp. 38–47 (cit. on p. 26).

[60] Andrea Gallidabino et al. "On the Architecture of Liquid Software: Technology Alternatives and Design Space". In: *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA16)*. IEEE. 2016, pp. 122–127 (cit. on pp. 11, 12, 164).

[61] Andrea Gallidabino et al. "Architecting Liquid Software". In: *Journal of Web Engineering* 16.5&6 (2017), pp. 433–470 (cit. on pp. 11, 12, 186).

[62] Andrea Gallidabino et al. "Liquid Web Applications: ICWE2017 Tutorial". In: *Proceedings of the International Conference on Web Engineering (ICWE17)*. Springer. 2017, pp. 269–271 (cit. on p. 13).

[63] Andrea Gallidabino. "Migrating and Pairing Recursive Stateful Components Between Multiple Devices with Liquid.js for Polymer". In: *Proceedings of the International Conference on Web Engineering (ICWE16)*. Springer, 2017, pp. 555–558 (cit. on p. 13).

[64] Andrea Gallidabino. "Liquid Web Architectures". In: *Proceedings of the International Conference on Web Engineering (ICWE19)*. Springer. 2019, pp. 560–565 (cit. on p. 13).

[65] Jessa James Garrett. *AJAX: A New Approach to Web Applications*. Archived from the original (http://www.adaptivepath.com/ideas/essays/archives/000385.php) to https://web.archive.org on 2 July. 2005 (cit. on pp. 35, 45).

[66] Andrea Gallidabino and Cesare Pautasso. "Deploying Stateful Web Components on Multiple Devices with Liquid.js for Polymer". In: *Proceedings of the International SIGSOFT Symposium on Component-Based Software Engineering (CBSE16)*. IEEE. 2016, pp. 85–90 (cit. on pp. 11, 12, 39, 69).

[67] Andrea Gallidabino and Cesare Pautasso. "The Liquid.js Framework for Migrating and Cloning Stateful Web Components across Multiple Devices". In: *Proceedings of the Companion to the International Conference on the World Wide Web (WWW16), Demonstrations*. 2016, pp. 183–186 (cit. on p. 13).

[68] Andrea Gallidabino and Cesare Pautasso. "Maturity Model for Liquid Web Architectures". In: *Proceedings of the International Conference on Web Engineering (ICWE17)*. Springer. 2017, pp. 206–224 (cit. on pp. 11, 12).

[69] Andrea Gallidabino and Cesare Pautasso. "Decentralized Computation Offloading on the Edge with Liquid WebWorkers". In: *Proceedings of the International Conference On Web Engineering (ICWE18)*. Springer. 2018, pp. 145–161 (cit. on pp. 12, 13).

[70] Andrea Gallidabino and Cesare Pautasso. "The Liquid User Experience API". In: *Proceedings of the Companion to The Web Conference 2018 (TheWebConf2018)*. 2018, pp. 767–774 (cit. on pp. 11, 12).

[71] Andrea Gallidabino and Cesare Pautasso. "The Liquid WebWorker API for Horizontal Offloading of Stateless Computations". In: *Journal of Web Engineering* 17.6 (2018), pp. 405–448 (cit. on pp. 12, 13, 158).

[72] Andrea Gallidabino and Cesare Pautasso. "Multi-Device Adaptation with Liquid Media Queries". In: *Proceedings of the International Conference On Web Engineering (ICWE19)*. Springer. 2019, pp. 474–489 (cit. on pp. 11, 12).

[73] Andrea Gallidabino and Cesare Pautasso. "Multi-Device Complementary View Adaptation with Liquid Media Queries". In: *Journal of Web Engineering* 18.8 (2020), pp. 1–40 (cit. on pp. 11, 12).

[74] Sergio Gusmeroli, Salvatore Piccione, and Domenico Rotondi. “A Capability-Based Security Approach to Manage Access Control in the Internet of Things”. In: *Mathematical and Computer Modelling* 58.5-6 (2013), pp. 1189–1205 (cit. on p. 164).

[75] Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. “On-Demand Cross-Device Interface Components Migration”. In: *Proceedings of the International Conference on Human Computer Interaction with Mobile Devices and Services*. ACM. 2010, pp. 299–308 (cit. on p. 17).

[76] Giuseppe Ghiani, Fabio Paternò, and Carmen Santoro. “Push and Pull of Web User Interfaces in Multi-Device Environments”. In: *Proceedings of the International Working Conference on Advanced Visual Interfaces (AVI12)*. ACM. 2012, pp. 10–17 (cit. on p. 24).

[77] Martin Gaedke and Jörn Rehse. “Supporting Compositional Reuse in Component-Based Web Engineering”. In: *Proceedings of the Symposium on Applied Computing*. ACM. 2000, pp. 927–933 (cit. on p. 26).

[78] Marko Gröönroos. *The Book of Vaadin, 4th Edition*. Vaadin Ltd, 2012 (cit. on p. 35).

[79] Jens Grubert et al. “Multifi: Multi Fidelity Interaction with Displays on and around the Body”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI15)*. ACM. 2015, pp. 3933–3942 (cit. on p. 25).

[80] Jonathan Grudin. “Computer-Supported Cooperative Work: History and Focus”. In: *Computer* 27.5 (1994), pp. 19–26 (cit. on p. 22).

[81] Martin Gaedke and Klaus Turowski. “Specification of Components Based on the Webcomposition Component Model”. In: *Data Warehousing and Web Engineering*. IGI Global, 2002, pp. 275–284 (cit. on p. 26).

[82] Dominique Guinard et al. “From the Internet of Things to the Web of Things: Resource-Oriented Architecture and Best Practices”. In: *Architecting the Internet of Things*. Springer, 2011, pp. 97–129 (cit. on p. 25).

[83] John Grundy, Xing Wang, and John Hosking. “Building Multi-Device, Component-Based, Thin-Client Groupware: Issues and Experiences”. In: *Proceedints of the Australian Computer Science Communications*. Vol. 24. Australian Computer Society, Inc. 2002, pp. 71–80 (cit. on p. 22).

[84] John Hartman et al. *Liquid Software: A New Paradigm for Networked Systems*. Tech. rep. 96-11. University of Arizona, 1996 (cit. on p. 17).

[85] John H. Hartman et al. “Joust: A Platform for Liquid Software”. In: *IEEE Computer* 32.4 (1999), pp. 50–56 (cit. on pp. 4, 17, 35, 39).

[86] Matias Hirsch et al. “Battery-Aware Centralized Schedulers for CPU-Bound Jobs in Mobile Grids”. In: *Pervasive and Mobile Computing* 29 (2016), pp. 73–94 (cit. on p. 96).

[87] Matias Hirsch et al. “A Two-Phase Energy-Aware Scheduling Approach for CPU-Intensive Jobs in Mobile Grids”. In: *Journal of Grid Computing* 15.1 (2017), pp. 55–80 (cit. on p. 28).

[88] Maria Husmann, Nicola Marcacci Rossi, and Moira C. Norrie. “Usage Analysis of Cross-Device Web Applications”. In: *Proceedings of the Symposium on Pervasive Displays*. ACM. 2016, pp. 212–219 (cit. on p. 5).

[89] Maria Husmann and Moira C Norrie. “XD-MVC: Support for Cross-Device Development”. In: *Proceedings of the International Workshop on Interacting with Multi-Device Ecologies in the Wild (Cross-Surface2015)*. ETH Zürich. 2015 (cit. on pp. 23, 39, 68, 69).

[90] Richard Han, Veronique Perret, and Mahmoud Naghshineh. “WebSplitter: a Unified XML Framework for Multi-Device Collaborative Web Browsing”. In: *Proceedings of the Conference on Computer Supported Cooperative Work*. ACM. 2000, pp. 221–230 (cit. on p. 5).

[91] Maria Husmann, Nicola Marcacci Rossi, and Moira C Norrie. “Usage Analysis of Cross-Device Web Applications”. In: *Proceedings of the International Symposium on Pervasive Displays (PerDis2016)*. ACM. 2016, pp. 212–219 (cit. on p. 3).

[92] Karl Huppler. “The Art of Building a Good Benchmark”. In: *Proceedings of the Technology Conference on Performance Evaluation and Benchmarking*. Springer. 2009, pp. 18–30 (cit. on p. 92).

[93] Maria Husmann et al. “UI Testing Cross-Device Applications”. In: *Proceedings of the International Conference on Interactive Surfaces and Spaces (ISS16)*. ACM. 2016, pp. 179–188 (cit. on p. 23).

[94] Mehdi Jazayeri. “Some Trends in Web Application Development”. In: *Proceedings of Future of Software Engineering, 2007 (FOSE'07)*. IEEE. 2007, pp. 199–213 (cit. on p. 56).

[95] Alan B Johnston and Daniel C Burnett. *WebRTC: APIs and RTCWEB protocols of the HTML5 real-time web*. Digital Codex LLC, 2012 (cit. on p. 165).

[96] David Johnson, Ntsibane Ntlatlapa, and Corinna Aichele. “Simple Pragmatic Approach to Mesh Routing Using BATMAN”. In: (2008) (cit. on p. 145).

[97] Tero Jokela, Jarno Ojala, and Thomas Olsson. “A Diary Study on Combining Multiple Information Devices in Everyday Activities and Tasks”. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI2015)*. ACM. 2015, pp. 3903–3912 (cit. on pp. 3, 110).

[98] Bettina Kemme and Gustavo Alonso. “Database Replication: A Tale of Research across Communities”. In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 5–12 (cit. on pp. 27, 40).

[99] Tim Kadlec. *Implementing Responsive Design: Building Sites for an Anywhere, Everywhere Web*. New Riders, 2012 (cit. on p. 106).

[100] Shaun Kane et al. “Exploring Cross-Device Web Use on PCs and Mobile Devices”. In: *Proceedings of IFIP Conference on Human-Computer Interaction*. Springer, 2009, pp. 722–735 (cit. on p. 4).

[101] Amy K Karlson et al. “Working Overtime: Patterns of Smartphone and PC Usage in the Day of an Information Worker”. In: *Proceedings of the International Conference on Pervasive Computing*. Springer. 2009, pp. 398–405 (cit. on p. 3).

[102] Fahim Kawsar and AJ Brush. “Home Computing Unplugged: Why, Where and When People Use Different Connected Devices at Home”. In: *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing (UbiComp2013)*. ACM. 2013, pp. 627–636 (cit. on pp. 3, 110, 160).

[103] Carel P Kruger and Gerhard P Hancke. “Benchmarking Internet of Things Devices”. In: *Proceedings of the International Conference on Industrial Informatics*. IEEE. 2014, pp. 611–616 (cit. on p. 92).

[104] Mikkel Baun Kjærgaard et al. “ndoor Positioning Using GPS Revisited”. In: *Pervasive Computing*. Springer, 2010, pp. 38–56 (cit. on p. 43).

[105] Karthik Kumar and Yung-Hsiang Lu. “Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?” In: *Computer* 43.4 (2010), pp. 51–56 (cit. on p. 25).

[106] Janne Kuuskeri, Janne Lautamäki, and Tommi Mikkonen. “Peer-to-Peer Collaboration in the Lively Kernel”. In: *Proceedings of the Symposium on Applied Computing*. 2010, pp. 812–817 (cit. on p. 40).

[107] Petr Knetl. *Complementary View Adaptation with Liquid.js*. Sept. 2019 (cit. on p. 160).

[108] Oskari Koskimies et al. “EDB: A Multi-Master Database for Liquid Multi-Device Software”. In: *Proceedings of the International Conference on Mobile Software Engineering and Systems*. IEEE. 2015, pp. 125–128 (cit. on pp. 34, 38, 53).

[109] Dejan Kovachev et al. “DireWolf - Distributing and Migrating User Interfaces for Widget-Based Web Applications”. In: *Proceedings of the International Conference on Web Engineering (ICWE13)*. Springer. 2013, pp. 99–113 (cit. on pp. 67, 69).

[110] Glenn E Krasner, Stephen T Pope, et al. “A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System”. In: *Journal of Object Oriented Programming* 1.3 (1988), pp. 26–49 (cit. on p. 136).

[111] Karthik Kumar et al. “A Survey of Computation Offloading for Mobile Systems”. In: *Mobile Networks and Applications* 18.1 (2013), pp. 129–140 (cit. on p. 85).

[112] Wei-Jenn Ke and Sheng-De Wang. “Reliability Evaluation for Distributed Computing Networks with Imperfect Nodes”. In: *IEEE Transactions on Reliability* 46.3 (1997), pp. 342–349 (cit. on p. 93).

[113] Michael Krug, Fabian Wiedemann, and Martin Gaedke. “Smartcomposition: A Component-Based Approach for Creating Multi-Screen Mashups”. In: *Proceedings of the International Conference on Web Engineering (ICWE14)*. Springer. 2014, pp. 236–253 (cit. on pp. 26, 67, 69).

[114] Marc Langheinrich. “Privacy by Design—Principles of Privacy-Aware Ubiquitous Systems”. In: *Proceedings of the International Conference on Ubiquitous Computing*. Springer. 2001, pp. 273–291 (cit. on pp. 162, 165).

[115] Kris Luyten and Karin Coninx. “Distributed User Interface Elements to Support Smart Interaction Spaces”. In: *Proceedings of the International Symposium on Multimedia*. IEEE. 2005 (cit. on p. 24).

[116] Michal Levin. *Designing Multi-Device Experiences: An Ecosystem Approach to User Experiences across Devices*. O’Reilly Media, Inc., 2014 (cit. on pp. 4, 17, 51).

[117] Hui Liu et al. “Survey of Wireless Indoor Positioning Techniques and Systems”. In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 37.6 (2007), pp. 1067–1080 (cit. on p. 43).

[118] Saadi Lahlou, Marc Langheinrich, and Carsten Röcker. “Privacy and Trust Issues with Invisible Computers”. In: *Communications of the ACM* 48.3 (2005), pp. 59–60 (cit. on p. 162).

[119] Avinash Lakshman and Prashant Malik. “Cassandra: a Decentralized Structured Storage System”. In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40 (cit. on p. 58).

[120] Seng W Loke et al. “Mobile Computations with Surrounding Devices: Proximity Sensing and Multilayered Work Stealing”. In: *ACM Transactions on Embedded Computing Systems* 14.2 (2015), p. 22 (cit. on p. 28).

[121] Avraham Leff and James T Rayfield. “Web-Application Development Using the Model/View/Controller Design Pattern”. In: *Proceedings of the International Conference on Enterprise Distributed Object Computing*. IEEE. 2001, pp. 118–127 (cit. on p. 44).

[122] Tom H Luan et al. “Fog Computing: Focusing on Mobile Users at the Edge”. In: *arXiv preprint arXiv:1502.01815* (2015) (cit. on p. 28).

[123] Yoël Luginbuhl. “Comparing Peer-to-Peer WebRTC Routing Strategies in Liquid.js”. MA thesis. USI, Feb. 2017 (cit. on p. 144).

[124] Essam Mansour et al. “A Demonstration of the Solid Platform for Social Web Applications”. In: *Proceedings of the International Conference Companion on World Wide Web*. 2016, pp. 223–226 (cit. on p. 21).

[125] Ethan Marcotte. *Responsive Web Design*. Editions Eyrolles, 2011 (cit. on pp. 6, 17, 51, 55, 105).

[126] Erik Meijer. “Democratizing the Cloud”. In: *Proceedings of the Companion to the Conference on Object-Oriented Programming Systems and Applications*. 2007, pp. 858–859 (cit. on pp. 35, 45).

[127] Giuseppe Mendola. “Roles and Groups for Access Control in Liquid Software”. MA thesis. USI, Sept. 2016 (cit. on p. 162).

[128] Takuya Maekawa, Takahiro Hara, and Shojiro Nishio. “A Collaborative Web Browsing System for Multiple Mobile Users”. In: *Proceedings of the International Conference on Pervasive Computing and Communications (PERCOM2006)*. IEEE. 2006, pp. 12–23 (cit. on p. 5).

[129] Nemanja Memarovic, Marc Langheinrich, and Florian Alt. “The Interacting Places Framework: Conceptualizing Public Display Applications that Promote cCommunity Interaction and Place Awareness”. In: *Proceedings of the International Symposium on Pervasive Displays*. 2012, pp. 1–6 (cit. on p. 26).

[130] Jonathan D Moffett. “Specification of Management Policies and Discretionary Access Control”. In: *Network and Distributed Systems Management* (1994), pp. 455–480 (cit. on p. 163).

[131] Michael S Mikowski and Josh C Powell. “Single Page Web Applications”. In: *B and W* (2013) (cit. on p. 45).

[132] Tommi Mikkonen, Kari Systä, and Cesare Pautasso. “Towards Liquid Web Applications”. In: *Proceedings of the International Conference on Web Engineering (ICWE15)*. Springer, 2015, pp. 134–143 (cit. on pp. 6, 17, 127, 163).

[133] Jonathan D Moffett, Morris Sloman, and Kevin P Twidle. “Specifying Discretionary Access Control Policy for Distributed Systems.” In: *Computer Communications* 13.9 (1990), pp. 571–580 (cit. on p. 164).

[134] Tommi Mikkonen and Antero Taivalsaari. “Cloud Computing and its Impact on Mobile Software Development: Two Roads Diverged”. In: *Journal of Systems and Software* 86.9 (2013), pp. 2318–2320 (cit. on pp. 6, 162).

[135] Jörg Müller et al. “Requirements and Design Space for Interactive Public Displays”. In: *Proceedings of the International Conference on Multimedia*. ACM. 2010, pp. 1285–1294 (cit. on p. 110).

[136] Bashar Nuseibeh and Steve Easterbrook. “Requirements Engineering: A Roadmap”. In: *Proceedings of the Conference on the Future of Software Engineering*. ACM. 2000, pp. 35–46 (cit. on p. 54).

[137] Michael Nebeling et al. “Interactive Development of Cross-Device User Interfaces”. In: *Proceedings of the International Conference on Human Factors in Computing Systems*. ACM. 2014, pp. 2793–2802 (cit. on pp. 16, 23).

[138] Qun Ni et al. “Privacy-Aware Role-Based Access Control”. In: *ACM Transactions on Information and System Security (TISSEC)* 13.3 (2010), pp. 1–31 (cit. on p. 164).

[139] Petru Nicolaescu et al. “Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types”. In: *Proceedings of the International Conference on Web Engineering (ICWE15)*. Springer, 2015, pp. 675–678 (cit. on pp. 128, 132, 152, 203).

[140] S Nandakumar and C Jennings. “SDP for the WebRTC”. In: (2012) (cit. on p. 176).

[141] Michael Nebeling, Maximilian Speicher, and Moira C Norrie. “CrowdAdapt: Enabling Crowdsourced Web Page Adaptation for Individual Viewing Conditions and Preferences”. In: *Proceedings of the SIGCHI Symposium on Engineering Interactive Computing Systems*. ACM. 2013, pp. 23–32 (cit. on pp. 66, 69).

[142] Jeff Offutt. “Quality Attributes of Web Software Applications”. In: *IEEE software* 19.2 (2002), pp. 25–32 (cit. on p. 54).

[143] Shusuke Okamoto and Masaki Kohana. “Load Distribution by Using Web Workers for a Real-Time Web Application”. In: *International Journal of Web Information Systems* 7.4 (2011), pp. 381–395 (cit. on p. 27).

[144] Tim O’reilly. *What is Web 2.0*. O’Reilly Media, Inc., 2009 (cit. on p. 105).

[145] Antti Oulasvirta and Lauri Sumari. “Mobile Kits and Laptop Trays: Managing Multiple Devices in Mobile Information Work”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2007, pp. 1127–1136 (cit. on pp. 3, 160).

[146] Antti Oulasvirta. “FEATURE When users" do" the Ubicomp”. In: *Interactions* 15.2 (2008), pp. 6–9 (cit. on p. 3).

[147] Evaggelia Pitoura and Bharat Bhargava. “Maintaining Consistency of Data in Mobile Distributed Environments”. In: *Proceedings of the International Conference on Distributed Computing Systems*. IEEE. 1995, pp. 404–413 (cit. on p. 27).

[148] Jorge Pérez et al. “Connectivity and Continuity: New Fronts in the Platform Wars”. In: *Panel at Twenty-First Americas Conference on Information Systems (AMCIS)*. 2015 (cit. on p. 27).

[149] Thomas D Palmer and N. Ann Fields. “Computer Supported Cooperative Work”. In: *Computer* 27.5 (1994), pp. 15–17 (cit. on p. 22).

[150] Stefan Poslad. *Ubiquitous Computing: Smart Devices, Environments and Interactions*. John Wiley & Sons, 2011 (cit. on pp. 3, 15, 28).

[151] Fabio Paternò and Carmen Santoro. “A Logical Framework for Multi-Device User Interfaces”. In: *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*. ACM. 2012, pp. 45–50 (cit. on p. 24).

[152] Soheil Qanbari et al. “IoT Design Patterns: Computational Constructs to Design, Build and Engineer Edge Applications”. In: *Proceedings of the International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE. 2016, pp. 277–282 (cit. on p. 4).

[153] Miguel Raposo and José Delgado. “Empowering the Web User with a Browserver”. In: *Proceedings of the International Conference on Enterprise Information Systems*. Springer. 2010, pp. 71–80 (cit. on pp. 67, 69).

[154] Eric Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Vol. 1. Addison-Wesley, 2001 (cit. on p. 176).

[155] Tom Rodden. “A Survey of CSCW Systems”. In: *Interacting with computers* 3.3 (1991), pp. 319–353 (cit. on p. 15).

[156] Umberto Sani. *Liquid User Experience*. Sept. 2016 (cit. on p. 242).

[157] Mahadev Satyanarayanan et al. “Pervasive Computing: Vision and Challenges”. In: *IEEE Personal Communications* 8.4 (2001), pp. 10–17 (cit. on p. 25).

[158] Mahadev Satyanarayanan et al. “Experience with Disconnected Operation in a Mobile Computing Environment”. In: *Mobile Computing*. Springer, 1993, pp. 537–570 (cit. on p. 93).

[159] Mahadev Satyanarayanan. “The Emergence of Edge Computing”. In: *Computer* 50.1 (2017), pp. 30–39. ISSN: 0018-9162 (cit. on p. 28).

[160] Bill Schilit, Norman Adams, and Roy Want. “Context-Aware Computing Applications”. In: *Proceedings of the Workshop on Mobile Computing Systems and Applications*. IEEE. 1994, pp. 85–90 (cit. on p. 26).

[161] Mark Stefik and John Seely Brown. “Toward portable ideas”. In: *Technological Support for Work Group Collaboration* (1989), pp. 147–165 (cit. on p. 22).

[162] Kjeld Schmidt and Liam Bannon. “Taking CSCW Seriously”. In: *Computer Supported Cooperative Work (CSCW)* 1.1-2 (1992), pp. 7–40 (cit. on p. 15).

[163] R. Schollmeier. “A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications”. In: *Proceedings of the International Conference on Peer-to-Peer Computing*. IEEE. 2001, pp. 101–102 (cit. on p. 68).

[164] Weisong Shi and Schahram Dustdar. “The Promise of Edge Computing”. In: *Computer* 49.5 (2016), pp. 78–81 (cit. on pp. 28, 164).

[165] Mary Shaw, David Garlan, et al. *Software Architecture*. Vol. 101. prentice Hall Englewood Cliffs, 1996 (cit. on p. 54).

[166] Steve Shafer et al. “The New EasyLiving Project at Microsoft Research”. In: *Proceedings of the Smart Spaces Workshop*. 1998, pp. 127–130 (cit. on p. 16).

[167] Swaminathan Sivasubramanian et al. “Analysis of Caching and Replication Strategies for Web Applications”. In: *IEEE Internet Computing* 11.1 (2007), pp. 60–66 (cit. on p. 58).

[168] Katarina Segerståhl and Harri Oinas-Kukkonen. “Distributed User Experience in Persuasive Technology Environments”. In: *Proceedings of the International Conference on Persuasive Technology*. Springer. 2007, pp. 80–91 (cit. on p. 4).

[169] Stephanie Santosa and Daniel Wigdor. “A Field Study of Multi-Device Workflows in Distributed Workspaces”. In: *Proceedings of the International Joint Conference on Pervasive and Ubiquitous Computing*. ACM. 2013, pp. 63–72 (cit. on p. 23).

[170] Antero Taivalsaari et al. “The Death of Binary Software: End User Software Moves to the Web”. In: *Proceedings of the International Conference on Creating, Connecting and Collaborating Through Computing*. IEEE. 2011, pp. 17–23 (cit. on p. 16).

[171] Mark Turner, David Budgen, and Pearl Brereton. “Turning Software into a Service”. In: *Computer* 36.10 (2003), pp. 38–44 (cit. on p. 30).

[172] A. Taivalsaari and T. Mikkonen. “A Roadmap to the Programmable World: Software Challenges in the IoT Era”. In: *IEEE Software* 34.1 (2017), pp. 72–80 (cit. on pp. 25, 70).

[173] Antero Taivalsaari, Tommi Mikkonen, and Kari Systä. “Cloud Browser: Enhancing the Web Browser with Cloud Sessions and Downloadable User Interface”. In: *Proceedings of the Conference on Grid and Pervasive Computing*. Springer. 2013, pp. 224–233 (cit. on p. 39).

[174] Antero Taivalsaari, Tommi Mikkonen, and Kari Systä. “Liquid Software Manifesto: The Era of Multiple Device Ownership and its Implications for Software Architecture”. In: *Proceedings of the Annual Computer Software and Applications Conference*. IEEE. 2014, pp. 338–343 (cit. on pp. 3, 15, 17–19, 31, 50, 53, 54).

[175] Vasileios Triglianos. *ASQ: Active Learning with Interactive Web Presentations and Classroom Analytics*. PhD Thesis at USI, 2018 (cit. on pp. 78, 102).

[176] Antero Taivalsaari and Kari Systä. “Cloudberry: An HTML5 Cloud Phone Platform for Mobile Devices”. In: *IEEE Software* 29.4 (2012), pp. 40–45 (cit. on pp. 30, 39, 48).

[177] Stefan Tilkov and Steve Vinoski. “Node.js: Using JavaScript to Build High-Performance Network Programs”. In: *IEEE Internet Computing* 14.6 (2010), pp. 80–83 (cit. on p. 57).

[178] Randika Upathilake, Yingkun Li, and Ashraf Matrawy. “A Classification of Web Browser Fingerprinting Techniques”. In: *Proceedings of the International Conference on New Technologies, Mobility and Security*. IEEE. 2015, pp. 1–5 (cit. on p. 97).

[179] Jari-Pekka Voutilainen, Tommi Mikkonen, and Kari Systä. “Synchronizing Application State Using Virtual DOM Trees”. In: *Proceedings of the International Workshop on Liquid Software*. Springer. 2016, pp. 142–154 (cit. on pp. 39, 67, 69).

[180] Jari-Pekka Voutilainen, Tommi Mikkonen, and Kari Systä. “Synchronizing Application State using Virtual DOM Trees”. In: *Proceedings of the International Conference on Web Engineering (ICWE16)*. Springer. 2016, pp. 142–154 (cit. on p. 4).

[181] Christian Vogt, Max Jonas Werner, and Thomas C Schmidt. “Leveraging WebRTC for P2P Content Distribution in Web Browsers”. In: *Proceedings of the International Conference on Network Protocols (ICNP13)*. IEEE. 2013, pp. 1–2 (cit. on pp. 41, 165).

[182] Andrew Whitmore, Anurag Agarwal, and Li Da Xu. “The Internet of Things - A Survey of Topics and Trends”. In: *Information Systems Frontiers* 17.2 (2015), pp. 261–274 (cit. on p. 25).

[183] Karl Wiegers and Joy Beatty. *Software Requirements*. Pearson Education, 2013 (cit. on p. 54).

[184] Mark Weiser. “The Computer for the 21st Century”. In: *Scientific American* 265.3 (1991), pp. 94–104 (cit. on p. 15).

[185] Mark Weiser. “Ubiquitous Computing”. In: *Proceedings of the ACM Conference on Computer Science*. Vol. 418. 10.1145. 1994, pp. 197530–197680 (cit. on p. 15).

[186] Evan Welbourne et al. “Building the Internet of Things Using RFID: the RFID Ecosystem Experience”. In: *IEEE Internet Computing* 13.3 (2009) (cit. on p. 25).

[187] Felix Wortmann and Kristina Flüchter. “Internet of Things”. In: *Business & Information Systems Engineering* 57.3 (2015), pp. 221–224 (cit. on p. 15).

[188] Mark Wallis, Frans Henskens, and Michael Hannaford. “A Distributed Content Storage Model for Web Applications”. In: *Proceedings of the International Conference on Evolving Internet*. IEEE. 2010, pp. 98–106 (cit. on p. 59).

[189] MH Willebeek-LeMair, Dilip D Kandlur, and Z-Y Shae. “On Multipoint Control Units for Videoconferencing”. In: *Proceedings of the conference on Local Computer Networks*. IEEE. 1994, pp. 356–364 (cit. on p. 146).

[190] Luke Welling and Laura Thomson. *PHP and MySQL Web Development*. Sams Publishing, 2003 (cit. on pp. 56, 57).

[191] Stelios Xinogalos, Kostas E Psannis, and Angelo Sifaleras. “Recent Advances Delivered by HTML5 in Mobile Cloud Computing Applications: A Survey”. In: *Proceedings of the Balkan Conference in Informatics*. ACM. 2012, pp. 199–204 (cit. on p. 27).

[192] Jishuo Yang and Daniel Wigdor. “Panelrama: Enabling Easy Specification of Cross-Device Web Applications”. In: *Proceedings of Annual ACM Conference on Human Factors in Computing Systems*. ACM. 2014, pp. 2783–2792 (cit. on pp. 67, 69).

[193] Andrea Zanella et al. “Internet of Things for Smart Cities”. In: *IEEE Internet of Things Journal* 1.1 (2014), pp. 22–32 (cit. on p. 25).

[194] Mikel Zorrilla et al. “A Web-Based Distributed Architecture for Multi-Device Adaptation in Media Applications”. In: *Personal and Ubiquitous Computing* 19.5-6 (2015), pp. 803–820 (cit. on p. 23).

Students

[Fis18] Alexander Fischer. *Liquid Web of Things*. Sept. 2018 (cit. on pp. 142, 255).

[Kne19] Petr Knetl. *Complementary View Adaptation with Liquid.js*. Sept. 2019 (cit. on p. 160).

[Lug17] Yoël Luginbuhl. "Comparing Peer-to-Peer WebRTC Routing Strategies in Liquid.js". MA thesis. USI, Feb. 2017 (cit. on p. 144).

[Men16] Giuseppe Mendola. "Roles and Groups for Access Control in Liquid Software". MA thesis. USI, Sept. 2016 (cit. on p. 162).

[San16] Umberto Sani. *Liquid User Experience*. Sept. 2016 (cit. on p. 242).

Web References

[IoT18] IoT Analytics, Knud Lasse Lueth. *State of the IoT 2018: Number of IoT Devices now at 7B – Market Accelerating*. 2018. URL: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/ (cit. on p. 3).

[Wor14] World Wide Web Foundation. *Web Index*. 2014. URL: https://thewebindex.org/report/ (cit. on p. 3).

[Wor19] World Wide Web Foundation. *New Mobile Broadband Pricing Data Shows Uneven Progress on Affordability*. 2019. URL: https://webfoundation.org/2019/03/new-mobile-broadband-pricing-data-reveals-stalling-progress-on-affordability/ (cit. on p. 3).

[Glo17] Global Connected Consumer Survey. *The Connected Consumer*. 2017. URL: http://www.google.com.sg/publicdata/explore?ds=dg8d1eetcqsb1_&hl=en&dl=en (cit. on pp. 3, 25, 70, 106).

[Goo17a] Google. *How Many Connected Devices Do People Use?* 2017. URL: https://www.consumerbarometer.com/en/graph-builder/?question=M3 (cit. on p. 3).

[Goo17b] Google. *Which Devices Do People Use?* 2017. URL: https://www.consumerbarometer.com/en/graph-builder/?question=M1 (cit. on p. 3).

[And15] Monica Anderson. *Technology Device Ownership: 2015*. 2015. URL: http://www.pewinternet.org/2015/10/29/technology-device-ownership-2015/ (cit. on p. 3).

[Goo12] Google. *The New Multi-Screen World: Understanding Cross-Platform Consumer Behavior*. 2012. URL: http://services.google.com/fh/files/misc/multiscreenworld_final.pdf (cit. on pp. 4, 127).

[Moz20a] Mozilla. *RTCDataChannel*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel (cit. on pp. 20, 74).

[Sho20] Shopify. *Liquid*. 2020. URL: https://shopify.github.io/liquid/ (cit. on p. 21).

[Eva11] Dave Evans. *The Internet of Things: How the Next Evolution of the Internet is Changing Everything*. 2011. URL: http://www.cisco.com/web/about/ac79/docs/innov/IoT (cit. on p. 25).

[Moz19a] Mozilla. *Web Components*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/Web_Components (cit. on p. 26).

[Pro20] Protocol Labs. *IPFS*. 2020. URL: https://ipfs.io/ (cit. on pp. 27, 75).

[Gal14] Gruman Galen. *Apple's Handoff: What works, and what doesn't*. 2014. URL: https://www.infoworld.com/article/2691101/apples-handoff-what-works-and-what-doesnt.html (cit. on pp. 29, 36, 39).

[Kar14] Bell Karissa. *Baton Promises to be the Ultimate Android App Switcher*. 2014. URL: https://mashable.com/2014/10/27/nextbit-baton-app/ (cit. on pp. 29, 39).

[Mic16] Microsft. *Continuum*. 2016. URL: https://www.microsoft.com/en-us/windows/continuum (cit. on pp. 29, 39).

[Goo20a] Google. *Chromecast*. 2020. URL: https://store.google.com/product/chromecast (cit. on p. 30).

[Ope09] Opera. *Opera Unite reinvents the Web*. 2009. URL: http://www.operasoftware.com/press/releases/general/opera-unite-reinvents-the-web (cit. on p. 30).

[Moz12] Mozilla. *Introduction to Firefox OS*. 2012. URL: https://developer.mozilla.org/en-US/docs/Archive/B2G_OS/Introduction (cit. on pp. 30, 48).

[App18] Apple Inc. *iCloud*. 2018. URL: https://www.icloud.com/ (cit. on pp. 30, 34).

[Goo20b] Google. *Google Sync*. 2020. URL: https://www.google.com/sync/index.html (cit. on pp. 30, 34).

[Rai16] Rails Core Team. *Ruby on Rails*. 2016. URL: https://rubyonrails.org/ (cit. on p. 35).

[Ora16] Oracle. *Sun Ray Products*. 2016. URL: http://www.oracle.com/technetwork/server-storage/sunrayproducts/overview/index.html (cit. on pp. 35, 36, 39, 44, 54, 64, 69).

[Goo20c] Google. *Firebase*. 2020. URL: https://firebase.google.com/ (cit. on p. 35).

[Goo16a] Google. *Google Web Toolkit*. 2016. URL: http://www.gwtproject.org/ (cit. on p. 35).

[Mic17] Microsoft. *Retro review: Microsoft's 2008 Surface Coffee Table in 2017*. 2017. URL: https://www.windowscentral.com/microsoft-surface-pixelsense-table (cit. on p. 43).

[Lin16] Linux Foundation. *Tizen Developers*. 2016. URL: https://developer.tizen.org/ (cit. on p. 48).

[Moz19b] Mozilla. *Using the application cache*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/HTML/Using_the_application_cache (cit. on p. 48).

[Moz20b] Mozilla. *Using Service Workers*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers (cit. on p. 48).

[DBE20] DB-Engines. *DB-Engines Ranking*. http://db-engines.com/en/ranking. 2020 (cit. on p. 56).

[W3C10] W3C. *Web SQL Database*. 2010. URL: https://www.oracle.com/database/ (cit. on p. 56).

[Moz19c] Mozilla. *IndexedDB API*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API (cit. on p. 56).

[Moz19d] Mozilla. *Window.localStorage*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage (cit. on pp. 56, 83).

[Moz19e] Mozilla. *Window.sessionStorage*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage (cit. on pp. 56, 83).

[Mic20] Microsoft. *ASP.NET*. 2020. URL: https://dotnet.microsoft.com/apps/aspnet (cit. on p. 57).

[Pou20] PouchDB. *Pouchdb*. 2020. URL: https://pouchdb.com/ (cit. on pp. 58, 189).

[Goo18] Google. *AngularJS*. 2018. URL: https://angularjs.org/ (cit. on p. 60).

[Fac20] Facebook Inc. *React*. 2020. URL: https://reactjs.org/ (cit. on p. 60).

[Til20] Tilde Inc. *Ember*. 2020. URL: https://emberjs.com/ (cit. on p. 60).

[Web17] Web Bluetooth Community Group. *Web Bluetooth*. 2017. URL: https://webbluetoothcg.github.io/web-bluetooth/ (cit. on pp. 70, 146, 256).

[Moz19f] Mozilla. *XMLHttpRequest*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest (cit. on p. 73).

[Moz19g] Mozilla. *Fetch API*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API (cit. on pp. 73, 86).

[Moz20c] Mozilla. *Promise*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise (cit. on pp. 73, 86, 227).

[Ope20] OpenJS foundation. *Node.js*. 2020. URL: https://nodejs.org/en/ (cit. on p. 74).

[WS 20] WS Community. *ws: a Node.js WebSocket Library*. 2020. URL: https://www.npmjs.com/package/ws (cit. on p. 74).

[Soc20] Socket.io Community. *socket.io*. 2020. URL: https://www.npmjs.com/package/socket.io (cit. on p. 74).

[Moz19h] Mozilla. *RTCPeerConnection*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection (cit. on p. 74).

[Ale20] Deveria Alexis. *Can I Usee WebRTC*. 2020. URL: https://caniuse.com/#search=webrtc (cit. on p. 74).

[Mic19] Bu Michelle. *PeerJS*. 2019. URL: https://peerjs.com/ (cit. on pp. 74, 133).

[Tom20] Cartwright Tom. *Socket.IO P2P*. 2020. URL: https://socket.io/blog/socket-io-p2p/ (cit. on p. 75).

[Fer20] Aboukhadijeh Feross. *WebTorrent*. 2020. URL: https://webtorrent.io/ (cit. on p. 75).

[Moz20d] Mozilla. *Using Web Workers*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers (cit. on pp. 85, 86).

[Moz19i] Mozilla. *Blob*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/Blob (cit. on pp. 87, 90).

[BD16] M Bynens and JD Dalton. *Benchmark.js v2.1.0*. 2016. URL: https://benchmarkjs.com/ (cit. on p. 93).

[Moz19j] Mozilla. *Battery Status API*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/Battery_Status_API (cit. on p. 97).

[Moz19k] Mozilla. *Network Information API*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/Network_Information_API (cit. on p. 98).

[W3C14] W3C. *WebComponents*. 2014. URL: https://www.w3.org/wiki/WebComponents/ (cit. on p. 128).

[The18a] The Polymer Project Authors. *Polymer Project*. 2018. URL: https://www.polymer-project.org/ (cit. on pp. 128, 136).

[Web20] WebComponent.org Community. *WebComponents.org*. 2020. URL: https://www.webcomponents.org/ (cit. on pp. 128, 224).

[Pol18] Polymer Project Authors. *Polymer Behaviors*. 2018. URL: https://polymer-library.polymer-project.org/3.0/docs/devguide/behaviors (cit. on p. 128).

[Ale19] Deveria Alex. *Can I Use*. 2019. URL: https://caniuse.com/ (cit. on p. 133).

[W3C16] W3C. *HTML Imports*. 2016. URL: http://www.w3.org/TR/html-imports/ (cit. on p. 135).

[The18b] The Polymer Project Authors. *About Polymer 1.0*. 2018. URL: https://polymer-library.polymer-project.org/1.0/docs/about_10 (cit. on p. 136).

[Moz20e] Mozilla. *Proxy*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy (cit. on p. 138).

[Moz19l] Mozilla. *Using shadow DOM*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_shadow_DOM (cit. on p. 142).

[Moz20f] Mozilla. *this*. 2020. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this (cit. on p. 142).

[Moz19m] Mozilla. *WebRTC Statistics API*. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_Statistics_API (cit. on p. 148).

[W3C19] W3C. *Media Queries Level 4*. 2019. URL: https://drafts.csswg.org/mediaqueries-4 (cit. on p. 153).

[Jac16] Spirou Jack. *clientJS*. 2016. URL: https://www.npmjs.com/package/clientjs (cit. on p. 187).

[The18c] The Polymer Project Authors. *paper-input*. 2018. URL: https://elements.polymer-project.org/elements/paper-input (cit. on p. 224).

[Goo16b] Google. *Google-map*. 2016. URL: https://www.webcomponents.org/element/GoogleWebComponents/google-map/1.2.0 (cit. on p. 230).

[Goo16c] Google. *Google-map-directions*. 2016. URL: https://www.webcomponents.org/element/GoogleWebComponents/google-map/1.2.0/google-map-directions (cit. on p. 230).

[Goo17c] Google. *Google-youtube*. 2017. URL: https://www.webcomponents.org/element/@google-web-components/google-youtube/2.0.0/google-youtube (cit. on pp. 235, 237).

[Wil20] Erik Wilde. *HTML5 Overview*. 2020. URL: http://html5-overview.net/ (cit. on p. 255).

Acronyms

AJAX Asynchronous JavaScript And XML. 35, 45, 73

API Application Programming Interface. x, xii, xvii, xxi, 11, 29, 31, 35, 57–59, 67, 73, 82, 83, 85–88, 91, 93, 97, 98, 128, 130–133, 135, 137, 138, 140, 142, 143, 146, 148–150, 155, 156, 158, 159, 165, 166, 170, 175, 180, 183–192, 243, 253, 255

B.A.T.M.A.N. Better Approach To Mobile Adhoc Networking. 145

BaaS Backend as a Service. 35

CPU Central Processing Unit. 8, 28, 44, 85, 98, 101, 212, 216, 253

CSCW Computer-Supported Collaborative Work. ix, 22–28

CSS Cascading Style Sheets. 64, 105, 106, 108, 122, 130, 135, 136, 142, 153, 224

CSS3 Cascading Style Sheets 3. 11, 105, 106, 108–110, 153, 154, 253

DAC Discretionary Access Control. xvii, 9, 13, 163–167, 171, 174, 254, 256

DOM Document Object Model. 67, 142, 182, 185, 228, 235

DUI Distributed User Interfaces. 7, 8, 24, 253

GPS Global Positioning System. 16, 43, 232, 233

GPU Graphics Processing Unit. 45

GUI Graphical User Interface. 23

GWT Google Web Toolkit. 35

HCI Human-Computer Interactions. ix, 24

HTML Hypertext Markup Language. 64, 130, 133, 135, 136, 142, 185, 224, 225, 227, 228, 238

HTML5 Hypertext Markup Language 5. v, 13, 16, 26, 27, 48, 57, 58, 60, 70, 73, 83, 85, 86, 91, 93, 97, 98, 106, 132, 133, 138, 164, 183, 251, 252, 255, 256

HTTP Hypertext Transfer Protocol. 55–57, 60, 61, 64, 65, 73, 86, 131

HTTPS Hypertext Transfer Protocol Secure. 176–179

IoT Internet of Things. v, ix, 3, 8, 20, 25, 28, 99, 106, 142, 164, 210, 255, 256

IP Internet Protocol. 43, 130, 165

IPFS InterPlanetary File System. 27, 75

JS JavaScript. 34, 83, 86, 87, 130, 132, 133, 135, 138, 149, 152, 164, 182, 184, 224

JSON JavaScript Object Notation. 138, 143, 147, 189, 210

LAN Local Area Network. 43

LPC LiquidPeerConnection. xvii, xxii, 144–150, 172, 193, 197, 199, 200, 202, 203, 205, 207, 210

LUE Liquid User Experience. vi, ix, x, xii, xv, xvii, xxi, 4, 6–8, 11, 19, 33–36, 38, 43, 46, 49–51, 54, 55, 60–63, 65, 66, 69, 70, 76, 82, 105, 108, 127, 134, 135, 137, 140–142, 145, 155, 158, 183, 185–187, 202, 235, 237, 238, 240, 241, 244, 245, 251–253, 255, 256

LWW Liquid WebWorker. x, xii, xvi, xviii, 13, 85–96, 98–102, 140, 150, 151, 183, 188, 212–216, 254

LWWPool Liquid WebWorker Pool. x, 86–92, 94–96, 140, 150–152, 212

MAC Media Access Control. 43

MCU Multipoint Control Unit. 146

MinCon Minimal Connection. 193, 195–203, 205–207, 209, 210

MVC Model-View-Controller. 44, 56, 57, 61, 70, 73

OLSR Optimized Link State Routing. 145

OS Operating System. 6, 46, 48, 52, 90, 212, 252

P2P Peer-to-Peer. xvii, xviii, xxi, 7, 27, 36, 38, 41, 49, 52, 55, 61, 68, 69, 74, 75, 84, 128, 130, 132, 133, 145, 155, 163, 164, 172, 176, 177, 179, 180, 191

PAN Personal Area Network. 43

PKG Message Packaging. 206, 209

QR Quick Response. 43, 44, 100

RQ Research Question. ix, xxi, 6–11, 252–254

RSSI Received Signal Strength Indication. 43

RSTP Secure Real-Time Protocol. 176

RT Routing Table. xii, xxii, 148, 149, 193, 197–203, 207–210

RTT Round Trip Time. 148

SaaS Software as a Service. 30

SDP Session Description Protocol. 176

SSID Service Set Identifier. 43

UI User Interface. vi, x, xi, xiii, xviii, xix, 4, 5, 8, 10, 11, 16–18, 22–24, 26, 29, 33–35, 45, 51, 52, 55, 57, 64, 67, 105–109, 111, 117–124, 128, 132, 134–136, 142, 161, 220, 222, 224, 228, 230, 231, 238, 240–245, 247, 249, 252, 253

URI Uniform Resource Identifier. xi, xxii, 60, 64, 82, 86, 90, 130, 131, 142–144, 149, 151, 152, 184, 188, 199, 202, 241, 242

URL Uniform Resource Locator. 43, 48, 55, 64, 65, 100

VM Virtual Machine. 46

W3C World Wide Web Consortium. 70, 132

WebRTC Web Real-Time Communication. 20, 56, 57, 61, 69, 74, 75, 84, 128, 130, 133, 142, 144, 146, 148, 163–165, 172, 175, 176, 180, 188, 191, 192, 194, 198, 199, 250, 256

WSS WebSocket Secure. 176, 177

XSL Extensible Stylesheet Language. 64