Journal of Web Engineering, Vol. 16, No. 5&6 (2017) 433–470
© Rinton Press

ARCHITECTING LIQUID SOFTWARE

ANDREA GALLIDABINO, CESARE PAUTASSO
Software Institute, Faculty of Informatics, Università della Svizzera Italiana
Via Buffi 13, Lugano, 6900 Switzerland
andrea.gallidabino@usi.ch, cesare.pautasso@usi.ch

TOMMI MIKKONEN
Department of Computer Science, University of Helsinki
Gustav Hällströmin katu 2b, FI-00560 Helsinki, Finland
tommi.mikkonen@helsinki.fi

KARI SYSTÄ, JARI-PEKKA VOUTILAINEN
Department of Pervasive Computing, Tampere University of Technology
Korkeakoulunkatu 1, FI-33720 Tampere, Finland
kari.systa@tut.fi, jari.voutilainen@iki.fi

ANTERO TAIVALSAARI
Nokia Technologies
Hatanpään valtatie 30, FI-33100 Tampere, Finland
antero.taivalsaari@nokia.com

The Liquid Software metaphor refers to software that can operate seamlessly across multiple devices owned by one or multiple users. Liquid Software applications can take advantage of the computing, storage and communication resources available on all the devices owned by the user. Liquid Software applications can also dynamically migrate from one device to another, following the user's attention and usage context. The key design goal in Liquid Software development is to minimize the additional efforts arising from multiple device ownership (e.g., installation, synchronization and general maintenance of personal computers, smartphones, tablets, home and car displays, and wearable devices), while keeping the users in full control of their devices, applications and data. In this paper we present the design space for Liquid Software, categorizing and discussing the most important architectural dimensions and technical choices. We also provide an introduction and comparison of two frameworks implementing Liquid Software capabilities in the context of the World Wide Web.

Keywords: Multi-device programming, multiple device ownership, software architecture, design space, Liquid Software.

1. Introduction

Device shipment trends [1] indicate that number of Web-enabled devices grows very rapidly. Every day, nearly four million new mobile devices, tablets and other types of connected devices are activated worldwide – well over five times more than the number of babies born each day. We are rapidly headed toward a future in which people own and use dozens of connected computing devices – laptops, phones, tablets, game consoles, TVs, car displays, digital photo frames, digital cameras, home appliances, watches, wearable computers, and so on. The users of these connected devices will expect interactive experiences with software designed to be immediately available, capable of delivering meaningful value even in few moments, without requiring active attention or explicit efforts dedicated to device management from the user's part, much in the fashion anticipated by Mark Weiser over twenty-five years ago [2].

In the era of multiple device ownership, software applications are no longer run only on personal computers but also on smartphones, tablets, phablets, smart TVs as well as in embedded devices found in houses, clothes and cars. Software usage patterns are changing accordingly, as the users increasingly assume the ability to access their data and applications on every applicable device, possibly even using many of those devices simultaneously [3]. However, the users are more and more exposed to the complexity that is caused by the large number of connected devices. This complexity arises, e.g., from the fact that user content is spread across several devices and services. Managing all the devices and services as separate entities is a tedious task; the situation gets much worse as the number of devices increases.

The fundamental problem in multiple device ownership is that traditional software applications and operating systems have not been designed to offer user experiences that would span multiple devices [4]. Instead, each device typically has its own set of applications that are installed and managed separately. Furthermore, all the devices have their own file systems and settings that need to be managed explicitly. The cost of managing applications, and ensuring that all the applications have access to all the relevant data files can become unbearable as the number of devices in a person's daily life grows.

At the same time, the users would simply want to focus on the task at hand, and use their devices as casually, effortlessly and efficiently as possible. Cloud-based systems such as Apple's iCloud[a], Google Sync[b] and Samsung Flow[c] are already paving the way for automatically synchronized devices. However, these systems are limited to devices supporting the same native ecosystem; in other words, they lock the users in a single vendor "silo". Furthermore, these systems do not yet provide seamless experiences and transitions across devices. Ideally, when the user moves 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 this paper we explore *Liquid Software* [5, 6] – a paradigm in which computation and user experience are expected to behave seamlessly across devices. Liquid Software applications are assumed to follow the user and have the ability to migrate and adapt to different usage contexts and device configurations. Liquid Software takes advantage of multiple heterogeneous devices, whereby devices can be used sequentially or simultaneously to run software that "roams" from one device to another, following the user's attention. Ideally, the user should be able to pick the best suited, most applicable device(s) for each situation, and use those devices for completing the tasks at hand.

[a] `http://www.icloud.com/`

[b] `http://www.google.com/sync/`

[c] `http://www.samsung.com/samsungflow/`

We will outline the central design issues and architectural dimensions and provide an architectural framework for building Liquid Software systems. We will first describe the concept of Liquid Software from the user perspective (Section 2), and connect it to emerging computing trends and related work (Section 3). We will then explore relevant architectural considerations (Section 4) and present the design space needed for implementing the concepts (Section 5). The resulting design space also provides a useful overview of emerging technologies and frameworks that support the implementation of Liquid Software.

The paper expands our previous work in the field of Liquid Software [7, 8, 9] by providing more detailed discussion and description of the architectural choices and dimensions. In addition, the paper includes additional contributions in the form of two Web frameworks for liquid Software – *Liquid.js for DOM* and *Liquid.js for Polymer* that are described and compared in Section 6. Towards the end of the paper, we draw some final conclusions in Section 7, and outline possible further research directions in Section 8.

2. Liquid Software User Experiences – Basic Use Cases and Scenarios

Designing applications that work seamlessly with a range of different devices – or even better, are able to adapt their behavior in accordance to their deployment context – requires special consideration in their design [3]. Broadly speaking, Liquid Software experiences can be divided into the following categories [10].

- **Sequential Use**. 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 needs in different usage contexts.
- **Simultaneous Use**. A single user uses the services from several devices at the same time, i.e., the session is open and running on multiple devices at same time. Different devices may show an adapted view of the same user interface, or the system may have a distributed user interface in which different devices play their own distinct roles.
- **Collaborative Use**. Several users run the same application on their devices. This collaboration can be either sequential or simultaneous.

Two basic Liquid Software usage scenarios are presented in Fig. 1. In the image on the left, the user is transferring a live application from one tablet to another (sequential collaborative scenario). In the image on the right, the user is transferring application state from her tablet to her car's navigation and entertainment system (sequential single-user scenario). In both cases, the assumption is that the users can continue doing what they were doing on one device on the other devices, with seamless, instantaneous and fluid transition between the devices, and taking advantage of the specific capabilities of each target device.

Two additional Liquid Software usage scenarios are presented in Fig. 2, depicting simultaneous, synchronized usage of multiple devices (this kind of behavior is also referred to as *device companionship*). In both of these scenarios, the user is leveraging the screen of a smaller device (in this case a mobile phone) as a control device for a larger device (in this case a tablet) so that the contents shown on the larger screen remain unobstructed by the users hands. Various similar use cases exist. For instance, nowadays it is already quite common to use a tablet device as a synchronized remote control device for an internet-connected TV.

The scenarios above share the same technical challenges in adapting the user interface to different devices and in synchronizing the data and state of the execution between devices.

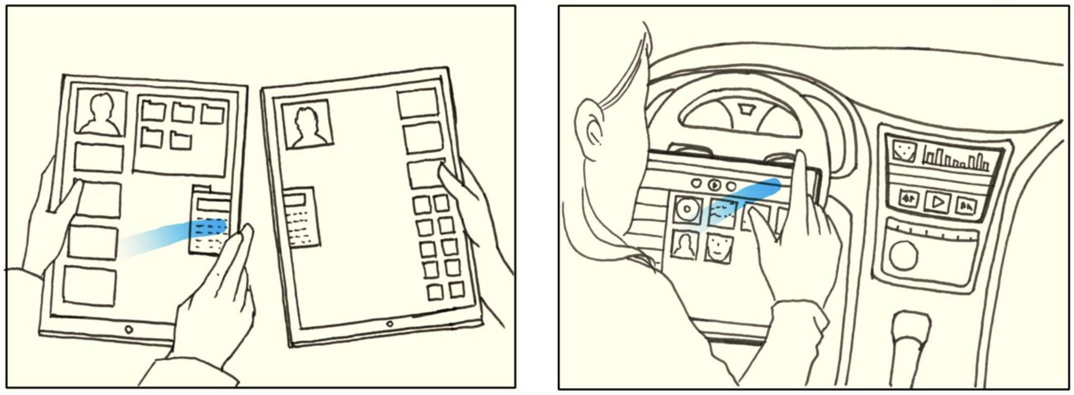

Fig. 1. Liquid Software Illustrated: Sequential Use Cases.

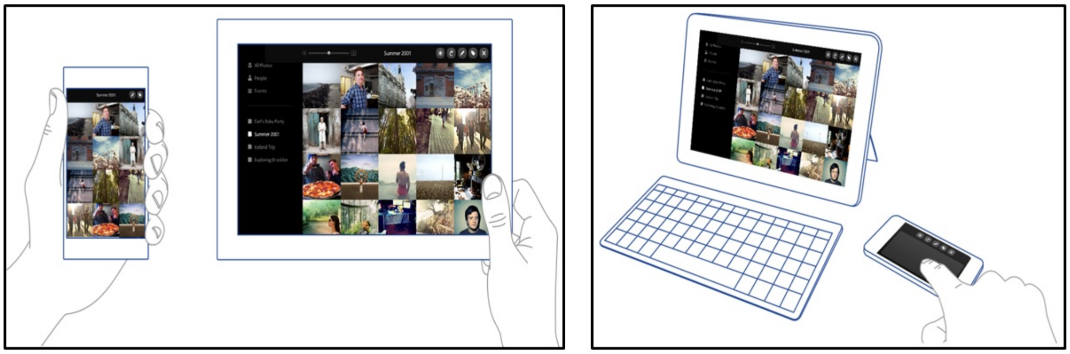

Fig. 2. Liquid Software Illustrated: Simultaneous Use Cases (Device Companionship).

Synchronization of the data and state are fundamental in the implementation of Liquid Software because the devices and users need to be aware of the results of their actions previously or simultaneously done in other devices. This is essential for transferring the work from one device to another, thus enabling seamless, real-time device usage.

Finally, a truly Liquid Software ecosystem should support heterogeneous devices across native software ecosystem boundaries. This way, developers would need to implement only one application, which could then adapt itself to run on various types of devices. A viable option in realizing such vision is to leverage the Web ecosystem – where applications already now are deployed on demand and through Responsive Web Design [11] are adapted to fit on the local device displays on the fly. 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 [12].

3. Background and Related Work

There have been numerous attempts to tackle the issues arising from multiple device ownership, with different design drivers. The very term *Liquid Software* was coined by Hartman, Manber, Peterson and Proebsting in a technical report back in 1996 [5]. Their seminal research culminated in the design of *Joust* [4] – a system that was based on synchronizing Java applications between virtual machines running in different computers.

Fluid computing [13] 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 to do. The authors list three main application areas: 1) multi-device applications, where several devices may be temporarily coupled to behave as one single device (for example, 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 coordination 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.

From the user interface perspective, the roots of Liquid Software can be traced back to *Computer-Supported Collaborative Work* (CSCW), where the focus is on enabling collaboration between multiple users rather than among the different devices owned by a single user [14]. A typical example of a collaborative, multi-device, component-based, thin client groupware system is presented in [15]. The design is based on Web technologies of the time, allowing incorporation of mobile devices as well as native clients in the same system. However, unlike in our view of Liquid Software, the approach focuses on groupware. Thus, while features associated with collaboration and multi-device use are built in, there is limited support for the seamless transfer of the actual state of the application.

Cloud computing systems, thin client environments and *Web-based applications* that adhere to the Software as a Service (SaaS) principles [16, 17] 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[d] or Cloudberry [18] explicit use of notification mechanisms (e.g., push notifications) is required at the application level if multiple clients are to be kept in sync.

In the wider area of mobile computing, the authors of [19] list various trends that can be related to Liquid Software. In the era of wearable computing, omnipresent connectivity, and increasingly smart devices, it is clear that techniques that enable seamless use of multiple devices will become fundamental. In addition to enabling a Liquid User Experience, such techniques can be used for offloading resource-intensive tasks to save the limited resources of mobile devices [20].

Responsive Web Design [11] is an established technique for adapting the user interface of applications to different devices capabilities. As in Liquid Software, the assumption is that

[d] https://developer.mozilla.org/docs/Mozilla/Firefox_OS

users may use different types of devices to access the application, with the user interface automatically adapting to the capabilities of the current device. However, in simultaneous or collaborative usage scenarios, it becomes important for a liquid Web application to adapt to the *set of devices* at the same time instead of just one device. Thus, Liquid Software pushes the boundaries for responsive UI capabilities even further.

Today, perhaps the most illustrative example of liquid Software behavior is the *Handoff* capability (also known as "Continuity" capability) in Apple's iOS ecosystem [21]. 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 user interface 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 [22]; 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 [23] 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[e] 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.

The cross-device user interfaces research area [24] proposes new techniques for enhancing cross-device interactions (see, e.g., Ctat [25]). Di Geronimo et al. propose new paradigms for interacting and sharing data between multiple mobile devices with intuitive gestures. Frameworks such as XD-MVC [26] provide developers with the information about the underlying hardware of each device in the system. Developers are encouraged to use the hardware infor-

[e] `https://www.google.com/chromecast/tv/?discover`

mation in order to divide pieces of the user interface of the application among a dynamic set of heterogeneous devices. The framework provides a *pattern library* that allows the application to be split across multiple devices following predefined patterns (e.g., controller-view pattern). The framework also allows the discovery of devices based on contact lists and the physical location of the devices [27]. In XD-MVC data synchronization occurs through a peer-to-peer network instead of relying on the cloud.

In our own work, we have built numerous experimental systems [28, 29, 30, 18, 31, 32, 33, 34, 35, 36, 37] to explore the boundaries of Liquid Software. However, there is still plenty of room for future experimentation and research prototypes both in the design space of Liquid Software as well as in understanding how the users perceive and use liquid applications. We have recently distilled the overall vision into the *Liquid Software Manifesto* [6], and described the main challenges in applying the vision to Web applications [7]. We have also introduced an architectural style for liquid Web services [30]. All of these techniques, following the principles laid out in [38], demonstrate how liquid applications can flow from computer to computer in a simple, straightforward fashion.

We will discuss the basic technical and architectural considerations in implementing Liquid Software in the next section.

4. Liquid Software – Essential Architectural Considerations

Liquid Software is by no means a single technology but rather a mindset for developing applications to be executed on multiple, heterogeneous computing devices [6]. Automatic synchronization of multiple computing devices today is at best supported only partially, usually within select ecosystems only, and even then the user must commonly turn it on explicitly. However, we believe that multiple device ownership will soon be so ubiquitous that automatic synchronization will become the norm rather than the exception. While Liquid Software requires a lot of underlying implementation effort, Liquid Software is really all about creating a seamless user *experience* that the underlying software environment or application must carefully nurture by following a number of principles and architectural considerations.

In the following, we discuss the most essential architectural considerations in individual subsections, covering adaptation, data and state migration as well as client vs. server application partitioning. These entail a number of key design decisions that form the design space of Liquid Software. We will dive into the design space in more detail in Section 5. Comments written in boldface within parentheses (**Example**) below refer to specific design dimensions that will be covered in more detail in Section 5.

4.1. *User Interface Adaptation to Different Devices and Contexts*

An essential characteristics of Liquid Software is its ability to adapt to take advantage of each and every device that it runs on either sequentially or in parallel (**Device Usage**).

The Liquid User Experience must consider and operate with the different *input methods* of the devices, such as the keyboard or the touch screen; for example, a small device might show only the most meaningful data as opposed to a device with a larger display. The use of *companion devices* makes responsiveness more challenging since the user interface needs to adapt to a combination of complementary devices. Thus, in general, the user interfaces of liquid applications should be responsive and adapt to the *set of* devices where they currently run (**UI Adaptation**).

While traditional, single-computer software expects the input and output channels to operate on the same device, Liquid Software does not have such restriction; liquid applications may well allow multiple input channels as well as multiple output channels from different devices. The users are free to use their devices as they please, making it possible to use the devices as comfortably as possible – for example by using the keyboard attached to one computer to type on the smartphone in which the liquid application has been deployed. In the simplest case, this could be achieved with basic input/output forwarding, without the need to actually migrate, fork or clone the software (**Primitives**).

4.2. *Data and State Synchronization*

In software migration it is important to distinguish between *persistent application data* and *dynamic application state* [38]. 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, persistent data – such as images and other content – is commonly stored locally on each device and can be synchronized using different cloud-based storage services [39]. 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 or Google Sync).

In addition to the persistent data consumed and produced by an application, Liquid Software is also concerned with the runtime (ephemeral, dynamic) state of the application. This runtime execution state can be captured at different levels of **granularity**, from the values of relevant variables (e.g., storing user interface configuration settings) or the entire volatile memory storage of an application. In traditional software applications, such data is not persisted after an application is switched off. However, in Liquid Software, the seamless operation of applications across devices requires the **Identification**, persistence, migration, **Replication** and **Synchronization** of such state in order to create a true sense of continuity to the user.

4.3. *Client/Server Partitioning*

When architecting liquid applications – in particular those implemented with Web technologies – the partitioning between the client and the server is a key design decision [40] (**Layering**). One extreme is to design applications so that they are run entirely on a backend server; the client is just a user interface that delegates the processing of all the events to the server. For instance, Ruby on Rails[f] is a prime example of this approach. Similar ultra thin techniques exist in the native realm where a virtual desktop is offered for remote use (e.g., in SunRay terminals [41]). In contrast with the server-centric view, there is the client-oriented approach where focus is placed on the software running on the client. Originating from Ajax [42], we today have single-page Web applications that use Backend as a Service (BaaS) APIs such as Firebase[g] deployed on a central server, providing persistent storage and notification

[f] `http://rubyonrails.org`

[g] `https://www.firebase.com/`

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 [4].

In practice, development can be both client- and server-centric, as many applications have both a rich client and separate server-side components. For instance, frameworks such as Vaadin [43] or Google Web Toolkit[h] allow the development of powerful user interfaces, while the focal point for developers is on the server side. In the context of liquid applications, the balance between centralization and decentralization can even change dynamically (**Topology**). 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.

4.4. *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. For instance, in the aforementioned 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 [41]. Likewise, when migrating applications to foreign devices, 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. An in-depth treatment of this matter falls beyond the scope of this paper.

5. The Design Space of Liquid Software

A Liquid User Experience (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 [21]. 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 peer-to-peer (P2P) connectivity for direct state synchronization across devices. The granularity of the software components that need to be migrated also impacts the Liquid User Experience, especially when deploying a liquid Web application over multiple devices that are intended to be used at the same time.

[h] `http://www.gwtproject.org/`

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 Fig. 3). 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 can be selected together (e.g., which Liquid User Experience **Primitives** are supported). We also indicate how the different alternatives constrain each other.

Table 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.

5.1. *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 peer-to-peer (P2P) fashion, leveraging local connectivity between devices. While peer-to-peer 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 as possible, too, with the cloud serving as an additional "peer", e.g., for backup purposes; this approach was used, e.g., in Nokia's EDB system [39].

5.1.1. *State Replication Topology*

In real-life implementations, the borderline between the two basic topologies is not always clear. For instance in [45], implementation techniques forced the design to use a central-

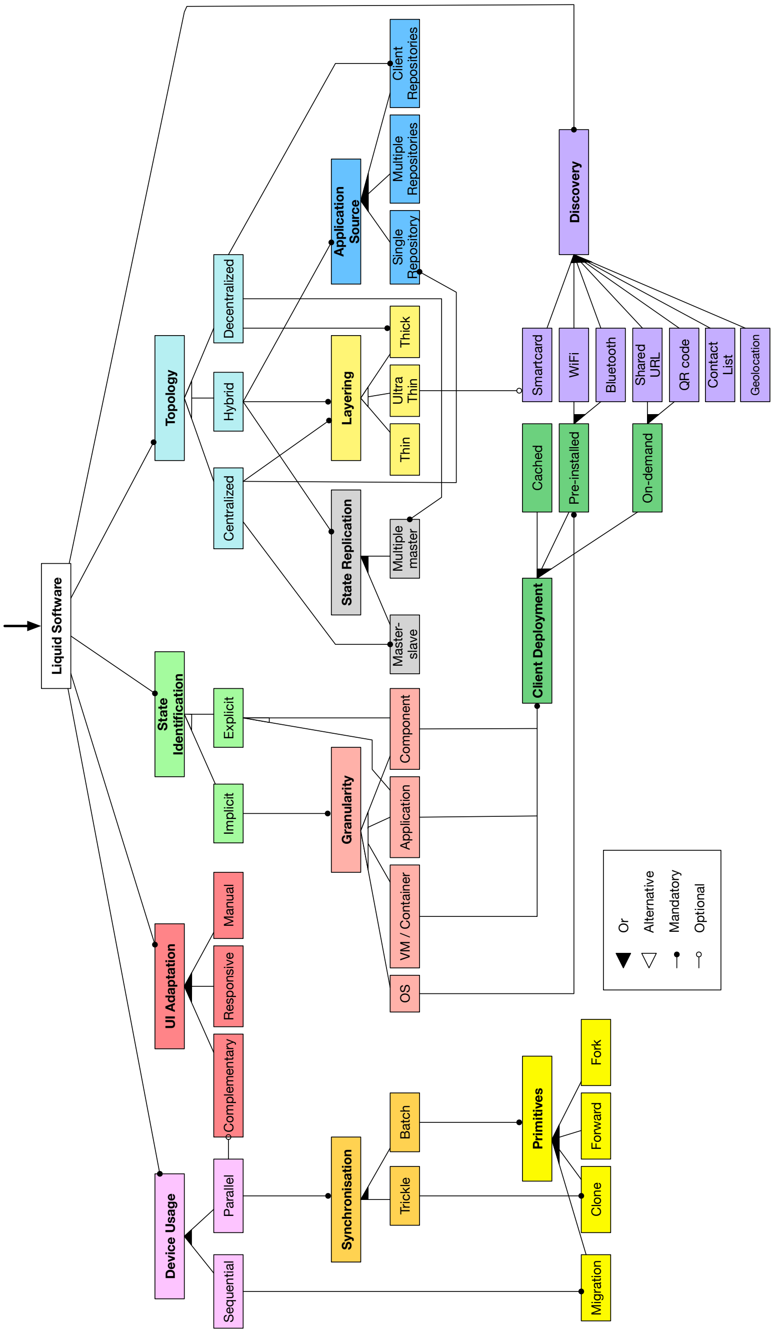

Fig. 3. Overview: the Design Space of Liquid Software. *Mandatory* arrows indicate that a child feature is required; *optional* arrows indicate that the child feature is optional; *alternative* arrows indicate that only one child feature must be selected; *or* arrows indicate that at least one child feature must be selected.

Table 1. Technologies Positioned in the Liquid Software Design Space

		Sun Ray [41]	Joust [4]	Fluid Computing [13]	Apple Continuity [21]	Android Baton[22]	Cloudberry [18] Cloudbrowser [44]	Continuum [23]	XD-MVC [26]	Liquid.js DOM [32]	Liquid.js for Polymer [33]
		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								✓		

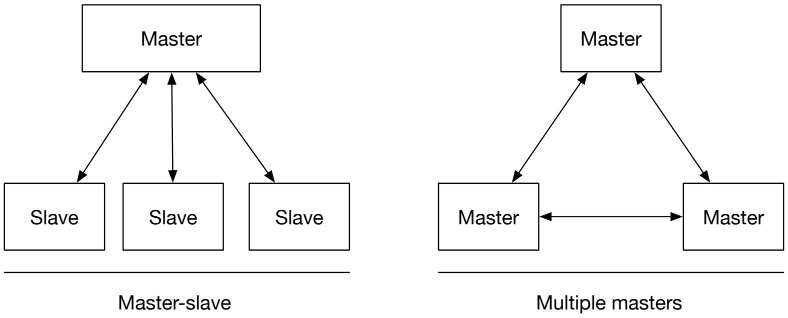

Fig. 4. State Replication Topology Alternatives

ized 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 Fig. 4) alternatives are:

- **Master-slave**: 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**: 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 [46].

5.1.2. *Application Source Topology*

In the area of application source topology (see Fig. 5) we recognize the following alternatives:

- **Single Repository**: 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**: the master copy of an application is stored in multiple nodes, such as multiple Web servers. In the multiple repository 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/propagaged to all the repositories (e.g., using a content

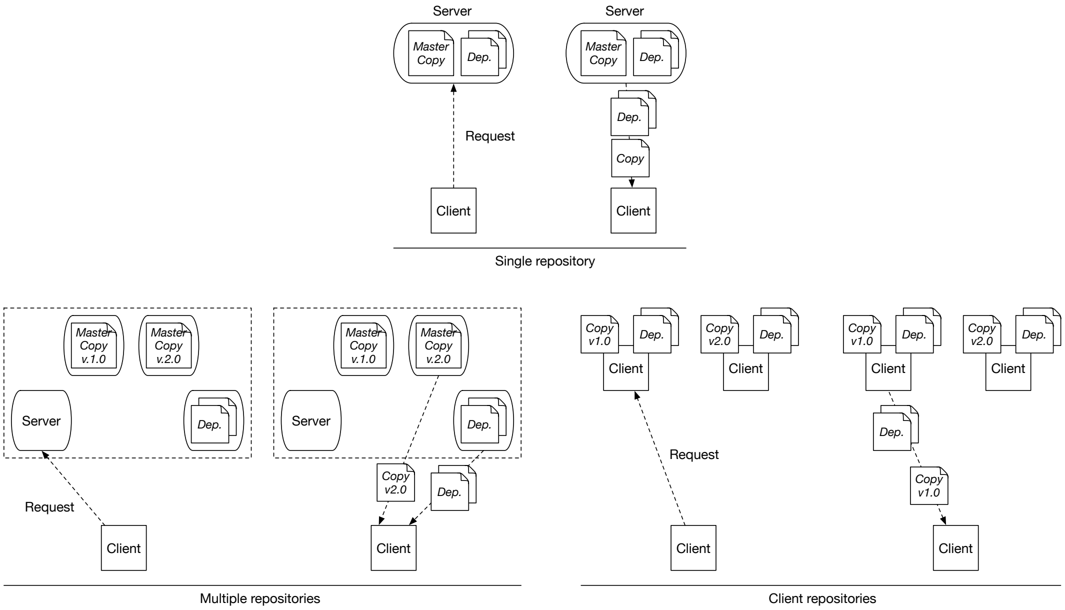

Fig. 5. Application Source Topology Alternatives

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**: 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 peer-to-peer communication [47]. 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.

5.2. *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 of devices is usually kept in the cloud.

On the device side, several technologies are readily available for discovery, including *shared URLs*, *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 Liquid User Experience. Many techniques can be employed in order to make the Liquid Software system aware of the available devices: by creating a Local Area Network or Personal Area Network 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 MAC addresses, while the 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 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 Liquid User Experience. 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 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 local area network technologies such as Wi-Fi or Bluetooth can easily compute the relative location of the connected device by using fingerprinting algorithms [48]. 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 RSSI-based approaches [49].

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.

5.3. *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 [50]. 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 – such as SunRay [41] – 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 [9]. In Fig. 6, we enumerate different logical layers of a Web application designed according to the Model-View-Controller (MVC) pattern [51]. 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 typical criteria [20] and tradeoffs 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 [42] and single-page applications [52] 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, 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 [40].

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.

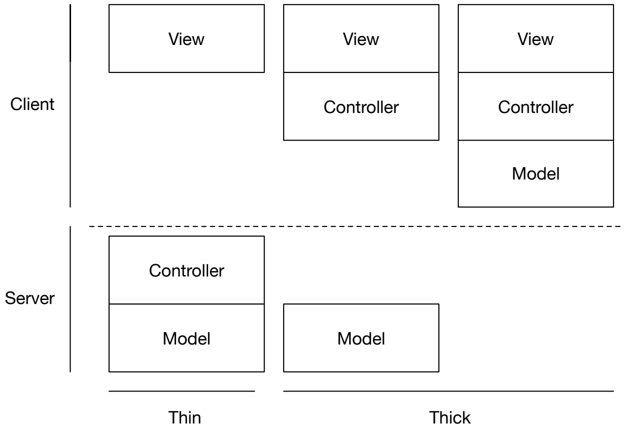

Fig. 6. Layering Alternatives [9]

Therefore, Liquid Software frameworks should offer capabilities for dynamically migrating computation from servers to clients and vice versa.

5.4. *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 (Fig. 7).

- *Operating system level.* The operating system and its underlying resources such as the file system, communication middleware and user interface follow the Liquid Software 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 operating system (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 liquid experience 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 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 virtual machines to run on the same physical resources to save energy. Like virtual machines, 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.

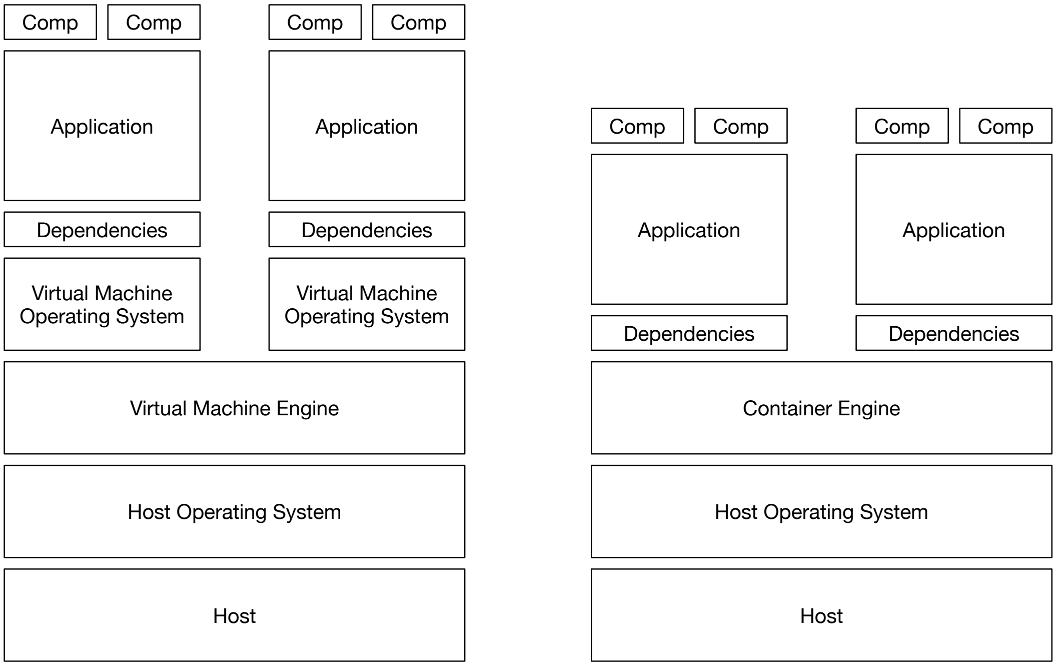

Fig. 7. Granularity Alternatives

- *Application level.* Moving a specific application as it is running is probably 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.

5.5. *Client Deployment*

There are numerous different ways to implement client software deployment (Fig. 9) and installation. In 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[i] and Firefox OS 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.

In 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 [18], applications were run by giving the URL to the Web engine; the application code was then *cached* using the HTML5 Application Cache [53]. 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.

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 (Fig. 8):

- *Thin client, on-demand deployment.* For thin client applications offline operation is not necessary and thus on-demand deployment is a feasible option.
- *Thin client, pre-installation.* 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.* One of the main 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 if Application Cache is used (at the time of writing the journal version of this paper, the HTML5 Application Cache mechanism was being deprecated from major Web browsers).
- *Thick client, pre-installation.* 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 peer-to-peer synchronization, the server might not be needed at all since everything is managed by the clients themselves.

5.6. *Liquid User Experience*

True Liquid User Experience 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.

[i] `https://developer.tizen.org/`

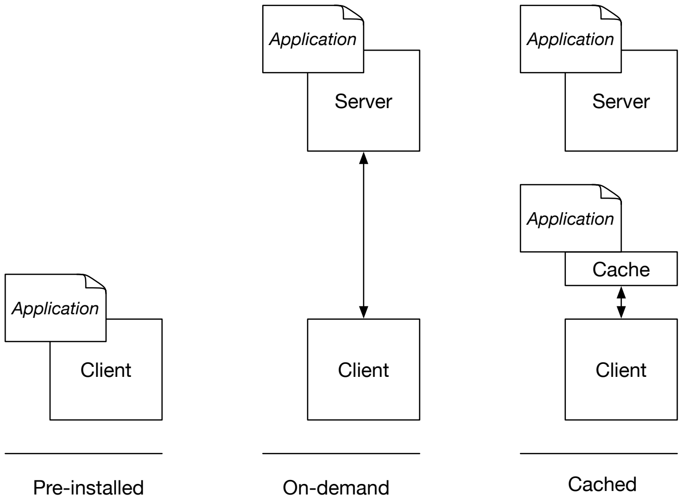

Fig. 8. Code Deployment alternatives

5.6.1. *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 reflecting the mobility of software [38] from one device to another, a combination of the following four primitives is used (Fig. 9):

- *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 [6], 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 example, 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

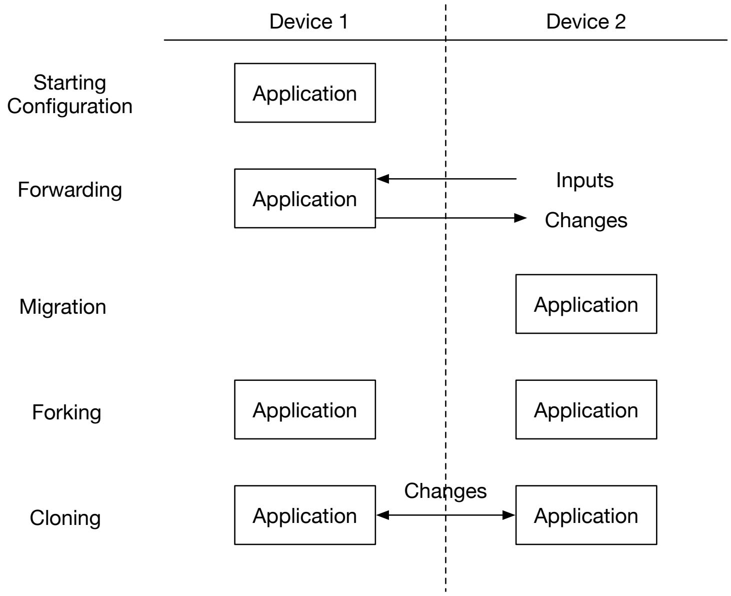

Fig. 9. Liquid User Experience Primitives

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.

5.6.2. *User Interface Adaptation*

There are different possible alternatives – *manual*, *responsive* and *complementary* – for deciding how to perform the user interface adaptation to the set of devices that are used for running the application [3]. With a manual approach, the users may directly activate the Liquid User Experience primitives to control how the user interface 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 user interface 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 [11] 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 user interface is adapted to each devices capabilities), but also a complementary user experience (where the capabilities of all the devices are fully exploited by the application with a distributed user interface).

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.

5.7. *State and Data in Liquid Software*

Broadly speaking, 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. Sequential screening – the user moving from one device to another to continue activities – does not generate conflicts, since there is only one active device at each time. In contrast, parallel and collaborative use of devices – when multiple devices are used simultaneously to complete a task – require close to real-time updates and may lead to conflicting updates to the same data. 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 user interface level, the trickle approach is pretty much 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 [54].

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 peer-to-peer 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 in [39] 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 [55] or the synchronization [56] 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.

5.8. *Security Considerations*

The success of computing platforms supporting liquid behavior is fundamentally dependent on *security*. As summarized in [6], 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 – most of the applications and data – 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 security 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 in [6] 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 [41] 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.

6. Web Frameworks for Liquid Software

As a part of this research we have independently composed two frameworks to study different design alternatives for Liquid Software. These frameworks are called *Liquid.js for DOM* and *Liquid.js for Polymer*. They have been created using the DOM manipulation capabilities and Web components[j] using the Polymer framework, respectively. The comparison here addresses the architectural design issues mentioned earlier in this paper, including topology and code deployment, data and state management, migration granularity, and user interface adaptation.

6.1. *Liquid.js for DOM*

The *Liquid.js for DOM* framework (we shall use the abbreviation *LfD* henceforth) provides easy-to-use mechanisms for migrating the DOM (Document Object Model) tree of the application to a different Web browser [32]. LfD is designed to automatically synchronize the contents of the DOM, and especially all the application state that is stored in the DOM. To synchronize the other parts of the local state the developer can register variables and functions to be synchronized together with the DOM. Figure 10 shows the features selected by Liquid.js for DOM with respect to the Feature model shown in Figure 3.

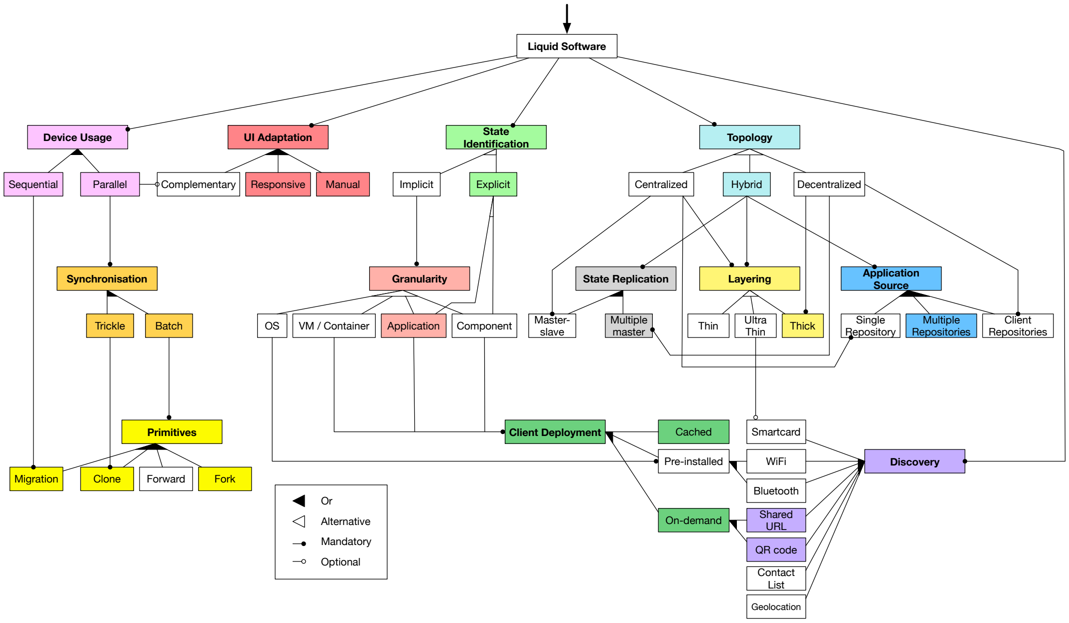

Fig. 10. Feature model choices for Liquid.js for DOM. The selected features are highlighted.

[j] `https://www.webcomponents.org/`

```
// initialization
var Liquid = require('liquid.js');
var liquid = new Liquid();

// Add data-handler="removeItem" to removeButton
removeButton.setAttribute('data-handler', 'removeItem');

// Register removeTodo event handler to bind
// click-event to DOM element with data-handler removeItem
liquid.registerHandler('removeItem', {'click': removeTodo});
```

Fig. 11. Initializing Liquid.js for DOM framework and registering functions for it.

6.1.1. *Overview*

LfD leverages *virtual DOM* technology that was originally designed for fast manipulation of DOM trees through an abstraction layer and popularized by React.js [57]. The framework is based on modern browser technologies, so that any desktop or mobile browser should work as long as the underlying technologies are supported. The basic idea of virtual DOM is to build an abstract version of the DOM tree instead of manipulating the DOM tree directly. This abstraction makes DOM manipulation significantly faster, especially when it is coupled with efficient comparison algorithms and operations on selected sub-trees. LfD is implemented as a JavaScript library that runs completely in the Web browser (on the client side), therefore requiring that it is included in the application. In addition, the application has to include some initialization code. After that the actual data migration can be automated. Since the virtual DOM can only handle data residing in the DOM, the initialization code must register variables and functions residing in JavaScript namespace to the framework for automatic migration. Example of initialization and function registeration is provided in Fig. 11.

6.1.2. *Topology and Code Deployment*

LfD is designed as a client-side framework, and thus it is deployed on an on-demand basis together with the application – it can be deployed even with a static Web page. The transparent caching mechanisms of the infrastructure have no effect on the deployment of the code. After deployment, the framework follows the thick client paradigm; in fact, it does not require a server at all.

LfD could be implemented both for a centralized and decentralized topology. In our proof-of-concept implementation, WebSockets are used for transferring the data, and this traffic flows through a centralized server. However, the library is implemented so that the communication mechanisms can be replaced practically with any communication technology. Thus the actual topology depends on the chosen transfer mechanism. Since LfD transfers only the state of the application, the code needs to come from a separate Web server.

6.1.3. *Granularity*

The entire application is always kept synchronized – it is not possible to keep one part of the application private and migrate only the other part. However, since only differences in the state are sent, the actual new content is limited to a subset of the DOM tree.

6.1.4. *Liquid User Experience*

LfD can support *migration*, *forking* and *cloning* from the Liquid User Experience primitives. LfD was designed with migration in mind; thus, it is implemented so that second device downloads the application from the server with the state transferred from another device. This state is applied to the initial state on the second application and the application on the first device can then be shut down. *Forking* is essentially similar except that the original browser continues its execution. *Cloning* can also be implemented – in that case the state synchronization is started.

6.1.5. *User Interface Adaptation*

Since LfD is directly based on DOM trees, it automatically supports DOM-based responsive user interfaces. For example, Bootstrap[k] and Foundation[l] libraries both implement their responsiveness using CSS classes; these classes are migrated with the DOM so they will work without any complications in different devices.

6.1.6. *Data and State*

The cornerstone of LfD is the initial state of the application. Once the framework is initialized, the initial state is stored as a virtual DOM tree within the framework. At any time the application is migrated, its current state is compared against the initial state, and the differences between the states are sent to another browser. The browser receiving the data will then apply the differences to its initial state in virtual DOM, and apply results to the actual DOM in the browser. The process is depicted in Fig. 12.

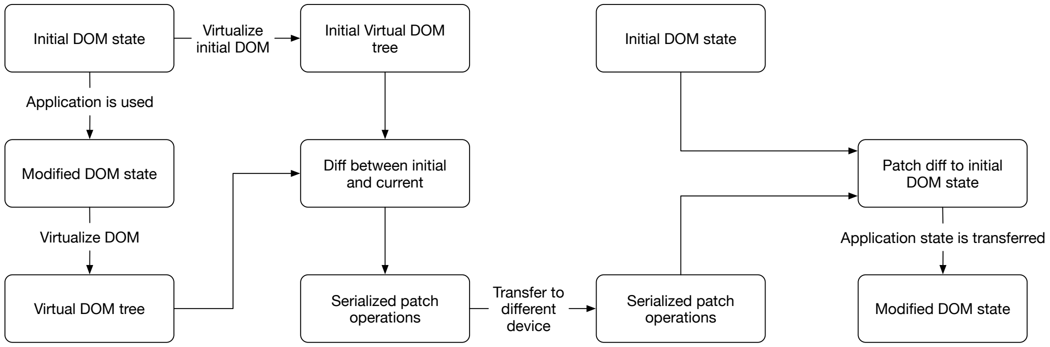
Initial DOM state
Virtualize initial DOM
Initial Virtual DOM tree
Initial DOM state
Application is used
Modified DOM state
Diff between initial and current
Patch diff to initial DOM state
Virtualize DOM
Application state is transferred
Virtual DOM tree
Serialized patch operations
Transfer to different device
Serialized patch operations
Modified DOM state

Fig. 12. Migrating the current DOM tree to another browser using Liquid.js for DOM.

If the application developer has declared relevant variables and functions that are migrated with the DOM tree, these are rebound after migration. Since everything in JavaScript namespace is not migrated, there might be some differences between different browsers in what the variables actually contain; it is left to the application developer to ensure that all the important variables of the application are migrated.

[k] http://getbootstrap.com/

[l] http://foundation.zurb.com/

The application developer can define when the synchronization is triggered. It may be triggered by some application specific event, or in the extreme case automatically in response to each and every change in the DOM tree. The choice depends on the application and required scenarios. Our design relies on the initial state, and therefore the old state in the receiving browser is always discarded during migration. This is a perfect match to *sequential use*. However, if the application is implemented to migrate after every state change immediately, *simultaneous use* can be supported rather well since the risk of conflicts is minimized.

Our framework supports both *trickle* and *batch* updates in synchronization. While comparing application state to the initial state, this in essence results in a batch update, since differences to the original state are transmitted. Trickle updates could be supported if the application is implemented so that migration is triggered when a change occurs in the application state. In practice migrations can be executed almost in real time. The application developer is in control how and how often the migrations will occur; this can range from the actions of the end user to specific browser events upon which the user has no control.

A live example of the LfD framework can be found in `https://liquidjs.herokuapp.com/` that defines a simple Todo application. The framework and the example are presented in detail in [32].

6.2. *Liquid.js for Polymer*

The Liquid.js for Polymer (LfP) framework enables the development of liquid Web applications developed with the Polymer framework – a framework developed by Google on top of Web Components [58]. Web Components allow the creation of reusable components in a Web application. LfP exploits the latest HTML5 standards, and thus applications behave correctly in all the Web browsers complying with them such as Google Chrome or Mozilla Firefox. The framework has been demonstrated in the WWW2016 and ICWE2016 conferences [34, 59]. More detailed information on the LfP API can be found in [33]. Figure 13 shows the features selected by Liquid.js for Polymer with respect to the Feature model shown in Figure 3.

6.2.1. *Overview*

LfP expects that applications are developed using a component-based approach, where the user interface of the Web application is composed out of one or more Web Components. The components of a Liquid application are built on top of the Polymer framework[m], with the injection of the liquid behavior and the addition of annotations provided by the LfP library to transform a Polymer Web component in a Liquid component.

LfP annotations define which components should manifest the Liquid User Experience 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 Fig. 14). Once the developers add their own annotations, LfP will transparently manage the deployment of the application as well as the state and data synchronization of a Liquid component; the framework also exposes APIs meant for the development of ad hoc Liquid User Experiences.

[m] `https://www.polymer-project.org/1.0/`

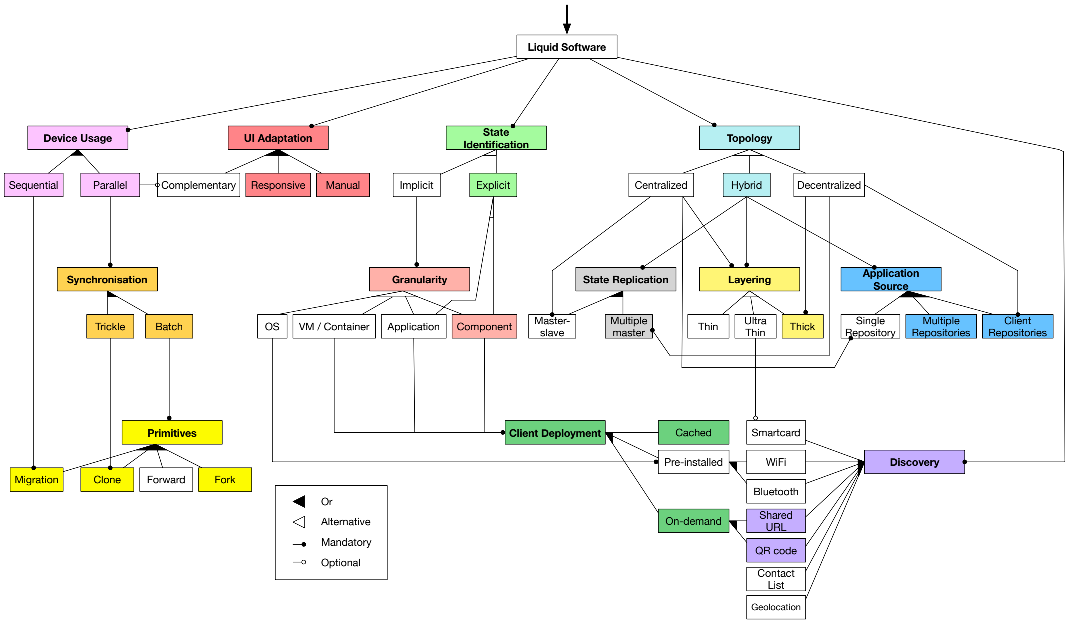

Fig. 13. Feature model choices for Liquid.js for Polymer. The selected features are highlighted.

LfP can be used for developing Liquid applications supporting both *sequential screening* and *parallel screening* scenarios, working either collaboratively with multiple users devices or among the devices of a single user.

More information about LfP (including a demo) can be found at `http://liquid.inf.usi.ch`. A demonstration of the tool and explanations on how to build a Liquid application can be found in [34].

6.2.2. *Topology and Code Deployment*

The LfP topology aims to be as *decentralized* as possible. Initially the master copy of the assets of an application resides 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 indexDB database of their Web browser. Afterwards any client can request another client to send their own copy of the application, which in turn can exchange it with anyone else. This approach allows LfP to take advantage of both *multiple repositories* and *client repository* as the source of the application (Fig. 15).

The exchange of assets happen through a Peer-to-Peer (P2P) mesh that is established dynamically at runtime. The P2P channels are created using the WebRTC's *RTCDataChannel* API available in modern Web browsers.

6.2.3. *Granularity*

The client of LfP is *thick*. Liquid Web applications developed in LfP are *component-based*

```
<dom-module id="liquid-component-example">
    <template>
        <!-- HTML here -->
    </template>
    <script>
        Polymer({
           is: 'liquid-component-example',
           behaviors: [LiquidBehavior], // Importing the behavior
           properties: {
                exampleProperty: {liquid: true} // Annotating a property
           },
        });
    </script>
</dom-module>
```

Fig. 14. Importing the LiquidBehavior and annotating properties

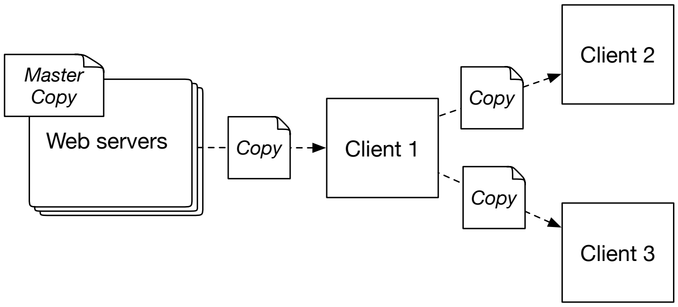

Fig. 15. LfP hybrid application deployment. Initially an application is stored in multiple Web servers. Clients that do not hold a copy of the application can request it from any server; when clients receive a copy from the servers they can start sharing it with other clients directly.

and all components can be shared between the set of devices. The client is responsible for transparently creating the P2P channels; 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 LfP 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.

6.2.4. *Liquid User Experience*

LfP exposes APIs that provide the mechanisms for creating the Liquid User Experience (LUE) the developers are looking for. LfP APIs give the developers the freedom to use LUE primitives directly, in particular LfP implements the *Migrate*, *Fork*, and *Clone* methods. It does not currently provide an implementation of the *Forwarding* method.

LfP also provides additional methods that allow developers to directly interact with four different levels of abstraction inside the framework (Fig. 16):

- **Device level**: the set of devices connected is the top level of abstraction of the framework, developers have access to the API and can use it to connect devices and send message among them at runtime. Lower levels of abstractions use this API automatically whenever API methods are called, but a developer may decide to use it directly if he so desires.
By introducing this level of abstraction it is possible to broadcast requests to all the devices; for example, it is possible to broadcast requests coming from a lower level of abstraction such as forking or cloning a component to all the devices.
The three primitives – migrate, fork and clone – can be applied at the device level. *Moving a device* refers to to taking all the components instantiated inside the device and moving them to another one; *forking a device* refers to instantiating a copy of all the components inside a device to another one; *cloning a device* refers to instantiating a copy of all the components inside a device to another one and synchronizing them for future state changes.
- **Assets level**: LfP exposes some methods to interact with the assets (static resources) of an application, such as requesting assets from a server or from another client directly. This API also allows caching and retrieval of assets in the indexDB database of the Web browser.
- **Liquid component level**: this level of abstraction allows the developer to instantiate components in a device. Any component that has a component model described in Polymer with the addition of LfP annotations can be used as a parameter of the set of methods exposed to this level of abstraction.
It is possible to apply the three Liquid primitives to the component instances: *migrate* a component somewhere else (on the same device but contained a different element in the DOM or on another device), *fork* a component: create a copy somewhere else, and *clone* a component: fork it and keep its state synchronized.
A Liquid component is the basic construction block of the Liquid application. Whenever a component is migrated, forked or cloned LfP takes care of sending both the state and the component model (asset) of the Liquid Component to the correct device and instantiating it. The model of a Liquid component is the definition of the component itself written with Polymer syntax and LfP annotations.
- **Liquid property level**: the annotations of LfP allow developers to define which properties are liquid. The framework exposes an API for pairing annotated properties directly between different components inside the same or different devices.

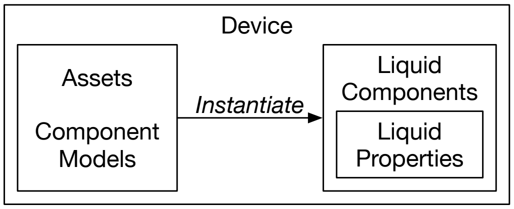

Fig. 16. The four levels of abstraction of Liquid.js for Polymer. Devices contain a set of assets and components. Components are instantiated from the Component Model described with Polymer and contain properties. A property is the state shared among components running on different devices and can be defined through the annotation included in the Component Model

Application created in LfP can adapt its user interface to different devices through CSS;

however, currently LfP does not provide any automated mechanism for adapting an application to the set of connected devices. Rather, the users are expected to *manually* drag and drop the application components on the devices they want to deploy them on. Additionally, LfP exposes an API that allows the developers to build their own policies, e.g., to recompute the layout of the application when some device disappears.

6.2.5. *State and Data in Liquid User Experience*

Data and state of an application are synchronized among the devices through the P2P channels described previously. The annotations of LfP make it possible to *explicitly* identify which parts of an applications should be exchanged and synchronized between devices. Data is usually stored in the clients of the applications using the *multiple masters* paradigm. However, developers are allowed to decide that data – which should be available to every single client connected – could be stored inside a Web server (thus shifting to *master-slave* paradigm for data replication). The same happens to the state of an application – the state is stored inside the clients and is directly exchanged between them through the P2P mesh. This can be accomplished with a special annotation on the Web Component that cannot be changed at runtime.

Ensuring consistency of data and state using a P2P architecture is a costly operation compared to a centralized approach, although it shifts the bandwidth usage and resource consumption from the server to the clients. Changes in the data and state are forwarded to all the clients as *trickle* updates, in which LfP framework sends incremental updates as soon as they are detected.

Finally, LfP allows clients to decide their own federation of synchronization with annotations. A client may decide to publish their own updates as well as subscribe to receive changes from other clients.

The state of a component is defined by its properties; an annotated property will be considered in the LfP framework a *Liquid Property.* Properties can be Javascript *booleans*, *strings*, *numbers*, *arrays* or *objects*; any of these properties can be annotated and LfP will manage them differently depending on their type. The framework will create a copy of the annotated properties and will ensure consistency whenever a property has to be synchronized among multiple devices using Yjs [60] (Fig. 17 and Fig. 18).

6.3. *Comparison*

Although LfD and LfP have been designed independently, there are many similarities between the frameworks. They are both client-based and both can use browser-to-browser communication, although some help from a central server is required, too.

The following characteristics distinguish the two frameworks:

- LfD focuses on the DOM, hence it targets the View of the Web application; LfP is designed to clearly separate between the View (rendered by each Web Component) and the Model (synchronized and migrated by the framework) of the Web application.

- LfP moves both code and state, while LfD moves only data. Thus, LfP supports more flexible architectures – even simple mobile agents could be developed with LfP.

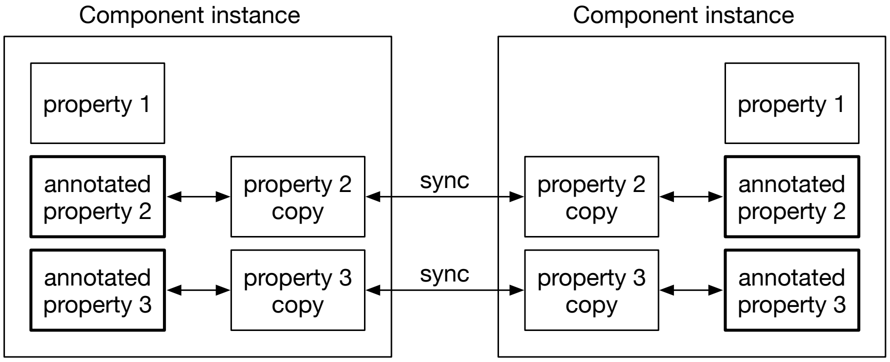

Fig. 17. In LfP every annotated property is copied, the framework will ensure that the state of the annotated property is consistent among all devices. The framework uses the copy as a proxy to detect live changes of the value of the property and fires event whenever one is detected.

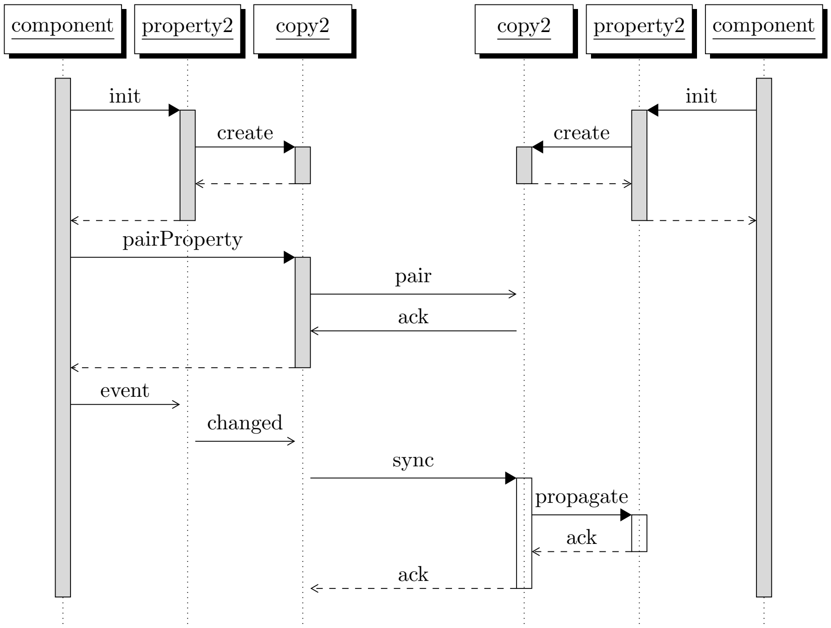

Fig. 18. Sequence diagram for property synchronization in LfP. The diagram is connected to the example shown in Fig. 17. When the component initializes it checks if a property is annotated. Whenever an annotated property is detected, a copy of it is created. If a property is paired and a change is detected, the property is synchronized automatically.

Table 2. Comparison

	Liquid.js for DOM (LfD)	Liquid.js for Polymer (LfP)
Layer	Thick (Server required only for deployment).	Thick (Model Synchronization).
Adaptation	Responsive Web Design.	Each component can adapt its UI to the device capabilities using Responsive Web Design techniques. Users need to manually choose on which device each component should run on.
Data migration	Only data that is located in the DOM tree is synchronized automatically; all other data must be registered with the framework for migration.	Fine-grained, explicit control over which data is migrated. Possible to migrate data directly between peers or also via the server.
State migration	Application level synchronization based on DOM; must be taken into account at design time for each application.	Component level synchronization based on Polymer properties; Fine-grained facilities explicitly allow programmers to control which component property is paired and where it should be stored.
Client deployment	Runs on client, but the application needs to be downloaded only once.	Runs on a client, but the application needs to be downloaded from the server only once. Creating the P2P communication channels with the WebRTC API requires a signaling server.
Discovery	Shared URL, also rendered as QR-code.	Shared URL, also rendered as QR-code.
Security	No authorization or authentication implemented yet; secure communications supported.	No authorization or authentication implemented yet; secure communications supported.

- LfP dictates the technology platform (Polymer/Web Components), while LfD is rather flexible (any Web browser).

- Concerning adaptation – LfD can use standard techniques such as Responsive Web Design, whereas LfP assumes that each Web component includes suitable adaptation mechanisms. On the other hand, LfP allows the developers to manually choose which components to deploy and run on each device.

- LfP provides the programmers means to control which component properties are liquid, wheres LfD does not as it synchronizes the content of the DOM.

- LfP provides means to define the master copy of the data, whereas LfD does not.

- LfP synchronizes state that is annotated explicitly (liquid properties of Web Components), whereas LfD synchronizes the entire DOM tree.

- LfD is focused on parallel, simultaneous device usage, with the ability to synchronize the DOM between two different Web browsers running at the same time. LfP supports both sequential and parallel usage scenarios, as it deals with both migration and synchronization of the liquid state of Web Components.

A detailed summary of similarities and differences between LfD and LfP are presented in Table 2. In addition to the features listed in the table, a key difference is that LfD is deep down based on the DOM, a data structure that is central in any Web application, whereas LfP builds on more recent technology that requires a component-based approach in the design of Web applications. While Web Components are supported by most modern browsers, they are not yet completely adopted by most Web applications. Both LfP and LfD use newly standardized features for browser-to-browser communication to circumvent the need for a central server that would host or relay the data. However, a server is still required for establishing the WebRTC communication channels between the browsers.

In general, we find that LfD can be helpful when aiming at enabling some liquid features in already existing applications, as it is directly building on the facilities that are at the very core of the browser. In contrast, LfP can be regarded as a more comprehensive Web application framework that builds on new technologies and can therefore provide more sophisticated support for building new liquid Web applications.

7. Conclusions

We take it for granted that we are at yet another turning point in the computing industry. The dominant era of PCs and smartphones is about to come to an end. So far, standalone devices have been the norm, and software has been primarily attached to a single device at a time. We believe that in the computing environment of the future, the users will have a considerably larger number of internet-connected devices in their daily lives.

The paradigm shift arising from ubiquitous multiple device ownership has inspired us to examine software architectures to enable what is known as Liquid Software. In this paper we have collected and presented the most important design decisions related to Liquid Software. We discussed the tradeoffs and characterized the impact, constraints and dependencies of each alternative. Moreover, we have identified different technology platforms for building Liquid Software, and specific Liquid Web Applications that fit in different niches in the presented design space.

As part of this work, we have constructed, presented and compared two Web frameworks for Liquid Software. Out of these two frameworks, Liquid.js for Polymer (LfP) provides more advanced support for building liquid component-based Web applications, while Liquid.js for DOM (LfD) allows the "liquification" of many existing Web applications with relatively little additional effort.

8. Future Work

There are plenty of potential avenues for further research in the area of Liquid Software. The presented design space includes numerous combinations that impact and constrain the resulting Liquid User Experience. Only some of these combinations have been explored so far. In the future the entire design space needs to explored in detail since there are many interesting combinations and tradeoffs, e.g., related to centralization and consistency vs. decentralization and privacy, as well as manual vs. automatic UI adaptation vs. usability and control.

The Web will keep playing a prominent role in the area Liquid Software, since the Web can act as a platform-independent execution environment that provides abstractions for serialization, migration, relocation, and adaptation that are needed for developing such applications. All the facilities already exist today, but require the use of special frameworks such as XD-MVC [26], Liquid.js for DOM [32], Liquid.js for Polymer [33], or even application-specific code. Ideally, the primitives required by Liquid Software should be standardized by bodies such as W3C, and be included in next-generation Web browsers as well as in native software development frameworks.

Especially security aspects require a lot more work; the challenge is to maximize ease of use and convenience while keeping the users in full control of their devices in a dynamic execution environment that may comprise both personal, shared and public devices. For example, an interesting area for further future work is to consider the use of public displays, or shared tabletop surfaces – where security related challenges require significantly more attention than in single-user, single-device cases.

One especially interesting area for future work for us is the emergence of the Internet of Things (IoT) as well as its Web-oriented counterpart, Web of Things (WoT [61]). In IoT and WoT solutions the number of computing units is usually dramatically larger than in traditional computing environments. For instance, within 10-15 years, our homes and offices can be expected to contain at least hundreds if not thousands of network-connected devices. The multiplication of connected devices in our surroundings will amplify the challenges associated with multiple device ownership. For instance, managing a very large number of computing units, storage and sensors/actuators that are not pre-allocated or pre-configured simply falls beyond what existing application models can handle. We believe that Liquid Software technologies are essential in constructing and modeling the behavior of software that is deployed in such complex, dynamic, heterogeneous, and pervasive environments.

Acknowledgments

This work has been supported by the Academy of Finland (project 295913). This work is also partially supported by the SNF and the Hasler Foundation with the Fundamentals of Parallel Programming for Platform-as-a-Service Clouds (SNF-200021_153560) and the Liquid Software Architecture (LiSA) grants. In addition, we wish to thank the individuals who have participated in the design and implementation of the different Liquid Software experiments.

References

1. Di Geronimo, L., Husmann, M., and Norrie, M. C. (2016) Surveying personal device ecosystems with cross-device applications in mind. *Proc. of the 5th ACM International Symposium on Per-*

vasive Displays, pp. 220–227, ACM.

2. Weiser, M. (1991) The computer for the 21st century. *Scientific American*, **265**, 94–104.
3. Levin, M. (2014) *Designing Multi-device Experiences: An Ecosystem Approach to User Experiences Across Devices*. O'Reilly.
4. Hartman, J. H., Bigot, P. A., Bridges, P. G., Montz, A. B., Piltz, R., Spatscheck, O., Proebsting, T. A., Peterson, L. L., and Bavier, A. C. (1999) Joust: A platform for liquid software. *IEEE Computer*, **32**, 50–56.
5. Hartman, J., Manber, U., Peterson, L., and Proebsting, T. (1996) Liquid software: A new paradigm for networked systems. Tech. Rep. 96-11, University of Arizona.
6. Taivalsaari, A., Mikkonen, T., and Systä, K. (2014) Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. *38th IEEE Computer Software and Applications Conference (COMPSAC)*, pp. 338–343, IEEE.
7. Mikkonen, T., Systä, K., and Pautasso, C. (2015) Towards liquid web applications. *Proc. of the 15th International Conference on Web Engineering*, pp. 134–143, Springer.
8. Gallidabino, A., Pautasso, C., Ilvonen, V., Mikkonen, T., Systä, K., Voutilainen, J.-P., and Taivalsaari, A. (2016) On the architecture of liquid software: Technology alternatives and design space. *Proc. of the 2016 Working IEEE/IFIP Conference on Software Architecture (WICSA 2016)*, pp. 122–127, IEEE.
9. Gallidabino, A. and Pautasso, C. (2017) Maturity model for liquid web architectures. *17th International Conference on Web Engineering (ICWE2017)*, Rome, Italy, June, pp. 206–224, Springer.
10. Google (2012), The new multi-screen world: Understanding cross-platform consumer behavior. `http://services.google.com/fh/files/misc/multiscreenworld_final.pdf`.
11. Marcotte, E. (2011) *Responsive Web Design*. Editions Eyrolles.
12. Mikkonen, T. and Taivalsaari, A. (2013) Cloud computing and its impact on mobile software development: Two roads diverged. *Journal of Systems and Software*, **86**, 2318–2320.
13. Bourges-Waldegg, D., Duponchel, Y., Graf, M., and Moser, M. (2005) The fluid computing middleware: Bringing application fluidity to the mobile internet. *IEEE/IPSJ International Symposium on Applications and the Internet (SAINT'05)*, pp. 54–63, IEEE.
14. Palmer, T. D. and Fields, N. A. (1994) Computer supported cooperative work. *Computer*, **27**, 15–17.
15. Grundy, J., Wang, X., and Hosking, J. (2002) Building multi-device, component-based, thin-client groupware: Issues and experiences. *Australian Computer Science Communications*, vol. 24, pp. 71–80, Australian Computer Society, Inc.
16. Turner, M., Budgen, D., and Brereton, P. (2003) Turning software into a service. *Computer*, **36**, 38–44.
17. Bouzid, A. and Rennyson, D. (2015) *The Art of SaaS: A Primer on the Fundamentals of Building and Running a Successful SaaS Business*. Xlibris.
18. Taivalsaari, A. and Systä, K. (2012) Cloudberry: An HTML5 cloud phone platform for mobile devices. *IEEE Software*, **29**, 40–45.
19. Picco, G. P., Julien, C., Murphy, A. L., Musolesi, M., and Roman, G.-C. (2014) Software engineering for mobility: Reflecting on the past, peering into the future. *Proc. of the on Future of Software Engineering*, pp. 13–28, ACM.
20. Dinh, H. T., Lee, C., Niyato, D., and Wang, P. (2013) A survey of mobile cloud computing: Architecture, applications, and approaches. *Wireless communications and mobile computing*, **13**, 1587–1611.
21. Gruman, G. (2014) Apple's Handoff: What works, and what doesn't. *InfoWorld*.
22. Bell, K. (2014) Baton promises to be the ultimate Android app switcher. *Mashable.com*.
23. Microsoft (2016), Microsoft Continuum. `http://www.windowscentral.com/continuum`.
24. Nebeling, M., Mintsi, T., Husmann, M., and Norrie, M. (2014) Interactive development of cross-device user interfaces. *Proc. of the 32nd annual ACM conference on Human factors in computing systems*, pp. 2793–2802, ACM.
25. Di Geronimo, L., Husmann, M., Patel, A., Tuerk, C., and Norrie, M. C. (2016) Ctat: Tilt-and-tap

across devices. *International Conference on Web Engineering*, pp. 96–113, Springer.

26. Husmann, M. and Norrie, M. C. (2015) XD-MVC: Support for cross-device development. *1st Intl. Workshop on Interacting with Multi-Device Ecologies in the Wild (Cross-Surface 2015)*, Zürich, ETH Zürich, Switzerland.
27. Husmann, M., Chithambaram, S., and Norrie, M. C. (2016) Combining physical and social proximity for device pairing. *Proc. of Cross Surface 2016*.
28. Kuuskeri, J. and Mikkonen, T. (2009) Partitioning web applications between the server and the client. *Proc. of the 2009 ACM Symposium on Applied Computing*, pp. 647–652, ACM.
29. Kuuskeri, J. and Mikkonen, T. (2010) Rest inspired code partitioning with a JavaScript middleware. *International Conference on Web Engineering*, pp. 244–255, Springer.
30. Bonetta, D. and Pautasso, C. (2011) An architectural style for liquid web services. *Proc. of the 2011 Working IEEE/IFIP Conference on Software Architecture (WICSA 2011)*, pp. 232–241, IEEE.
31. Systä, K., Mikkonen, T., and Järvenpää, L. (2013) HTML5 agents – mobile agents for the Web. *Proc. of the International Conference on Web Information Systems and Technologies (WEBIST)*, pp. 37–44, SciTePress.
32. Voutilainen, J.-P., Mikkonen, T., and Systä, K. (2016) Synchronizing application state using virtual DOM trees. *Proc. of the 1st International Workshop on Liquid Software*, pp. 142–154, Springer.
33. Gallidabino, A. and Pautasso, C. (2016) Deploying stateful web components on multiple devices with Liquid.js for Polymer. *Proc. of 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE*, pp. 85–90, IEEE.
34. Gallidabino, A. and Pautasso, C. (2016) The Liquid.js framework for migrating and cloning stateful web components across multiple devices. *Proc. of the 25th International Conference on the World Wide Web (WWW), Demonstrations*, pp. 183–186, International World Wide Web Conferences Steering Committee.
35. Babazadeh, M., Gallidabino, A., and Pautasso, C. (2015) Liquid stream processing across web browsers and web servers. *Proc. of the 15th International Conference on Web Engineering (ICWE2015)*, Rotterdam, NL, June, pp. 24–33, Springer.
36. Triglianos, V. and Pautasso, C. (2015) Asqium: A JavaScript plugin framework for extensible client and server-side components. *Engineering the Web in the Big Data Era*, pp. 81–98, Springer.
37. Mäkitalo, N., Pääkkö, J., Raatikainen, M., Myllärniemi, V., Aaltonen, T., Leppänen, T., Männistö, T., and Mikkonen, T. (2012) Social devices: Collaborative co-located interactions in a mobile cloud. *Proc. of the 11th International Conference on Mobile and Ubiquitous Multimedia*, p. 10, ACM.
38. Fuggetta, A., Picco, G. P., and Vigna, G. (1998) Understanding code mobility. *IEEE Trans. Softw. Eng.*, **24**, 342–361.
39. Koskimies, O., Wikman, J., Mikola, T., and Taivalsaari, A. (2015) EDB: A multi-master database for liquid multi-device software. *Proc. of the Second ACM International Conference on Mobile Software Engineering and Systems*, pp. 125–128, IEEE.
40. Meijer, E. (2007) Democratizing the cloud. *Proc. of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA'07)*, pp. 858–859, ACM.
41. Oracle (2016), Sun Ray products. `http://www.oracle.com/technetwork/server-storage/sunrayproducts/overview/index.html`.
42. Garrett, J. J., 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.
43. Grönroos, M. (2012) *The Book of Vaadin, 4th Edition*. Vaadin Ltd.
44. Taivalsaari, A., Mikkonen, T., and Systä, K. (2013) Cloud browser: Enhancing the web browser with cloud sessions and downloadable user interface. *Grid and Pervasive Computing*, pp. 224–233, Springer.
45. Kuuskeri, J., Lautamäki, J., and Mikkonen, T. (2010) Peer-to-peer collaboration in the Lively Kernel. *Proc. of the ACM Symposium on Applied Computing*, pp. 812–817, ACM.

46. Kemme, B. and Alonso, G. (2010) Database replication: A tale of research across communities. *Proc. of the VLDB Endowment*, **3**, 5–12.
47. Vogt, C., Werner, M. J., and Schmidt, T. C. (2013) Leveraging WebRTC for P2P content distribution in web browsers. *Proc. of the 21st IEEE International Conference on Network Protocols (ICNP 2013)*, pp. 1–2, IEEE.
48. Liu, H., Darabi, H., Banerjee, P., and Liu, J. (2007) Survey of wireless indoor positioning techniques and systems. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, **37**, 1067–1080.
49. Kjærgaard, M. B., Blunck, H., Godsk, T., Toftkjær, T., Christensen, D. L., and Grønbæk, K. (2010) Indoor positioning using GPS revisited. *Pervasive Computing*, pp. 38–56, Springer.
50. Casteleyn, S., Garrigós, I., and Mazón, J.-N. (2014) Ten years of Rich Internet Applications: A systematic mapping study, and beyond. *ACM Trans. Web*, **8**, 18:1–18:46.
51. Leff, A. and Rayfield, J. T. (2001) Web-application development using the model/view/controller design pattern. *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pp. 118–127, IEEE.
52. Mikowski, M. S. and Powell, J. C. (2013) Single page web applications. *B and W*.
53. `w3schools.com` (2016), HTML5 Application Cache. `https://www.w3schools.com/html/html5_app_cache.asp`.
54. Brewer, E. (2012) Cap twelve years later: How the "rules" have changed. *Computer*, **45**, 23–29.
55. Bradford, R., Kotsovinos, E., Feldmann, A., and Schiöberg, H. (2007) Live wide-area migration of virtual machines including local persistent state. *Proc. of the 3rd international conference on Virtual execution environments*, pp. 169–179, ACM.
56. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. (2008) Remus: High availability via asynchronous virtual machine replication. *Proc. of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pp. 161–174, San Francisco.
57. Facebook (2016), React: A JavaScript library for building user interfaces. `http://facebook.github.io/react/`.
58. Savage, T. (2015) Componentizing the Web. *Communications of the ACM*, **58**, 55–61.
59. Gallidabino, A. (2016) Migrating and pairing recursive stateful components between multiple devices with Liquid.js for Polymer. *Proc. of the 16th International Conference on Web Engineering (ICWE2016)*, pp. 555–558, Springer.
60. Nicolaescu, P., Jahns, K., Derntl, M., and Klamma, R. (2016) Near real-time peer-to-peer shared editing on extensible data types. *Proceedings of the 19th International Conference on Supporting Group Work*, pp. 39–49, ACM.
61. Guinard, D., Trifa, V., Mattern, F., and Wilde, E. (2011) From the Internet of Things to the Web of Things: Resource-oriented architecture and best practices. Uckelmann, D., Harrison, M., and Michahelles, F. (eds.), *Architecting the Internet of Things*, pp. 97–129, Springer.