On the Architecture of Liquid Software: Technology Alternatives and Design Space

Andrea Gallidabino and Cesare Pautasso
University of Lugano (USI)
CH-6900 Lugano, Switzerland
andrea.gallidabino@usi.ch, c.pautasso@ieee.org

Ville Ilvonen, Tommi Mikkonen, Kari Systä and Jari-Pekka Voutilainen
Tampere University of Technology
FI-33720 Tampere, Finland
ville.ilvonen@student.tut.fi, tommi.mikkonen@tut.fi, kari.systa@tut.fi, jari.voutilainen@iki.fi

Antero Taivalsaari
Nokia Technologies
Visiokatu 1
FI-33720 Tampere, Finland
antero.taivalsaari@nokia.com

***Abstract*—The *liquid* metaphor refers to software that operates seamlessly across multiple devices owned by one or multiple users. Liquid software architectures can dynamically deploy and redeploy stateful software components and transparently adapt them to the capabilities of heterogeneous target devices. The key design goal in liquid software development is to minimize the efforts that are related to multiple device ownership (e.g., installation, synchronization and general maintenance of personal computers, smartphones, tablets, home displays, cars and wearable devices), while keeping the users in full control of their devices, applications and data. In this paper we present a design space for liquid software, categorizing and discussing the most important architectural issues and alternatives. These alternatives represent relevant capabilities offered by emerging technologies and deployment platforms that are then positioned and compared within the design space presented in the paper.**

***Keywords*—Multi-device programming, multiple device ownership, software architecture, design space, liquid software.**

I. Introduction

We are rapidly heading toward a future in which the users own a large number of network-connected computing devices. 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 expect to be able to access their data and applications seamlessly on every device, possibly even using those devices at the same time [1].

Traditional software and operating systems are not designed to offer user experiences that take advantage of multiple devices [2]. Instead, each device has its own set of applications, installed and managed separately. However, the cost of managing applications and ensuring that all applications have access to all the relevant data can easily become unbearable as the number of devices in a person's daily life grows. In this paper we explore *liquid software* [3] – a paradigm in which computation and user experience are expected to behave seamlessly across devices. Applications can migrate and adapt to different usage contexts and device configurations. Liquid software takes full advantage of multiple heterogeneous devices, whereby any device can be used sequentially (or concurrently) to run software that "roams" from one device to another, following the user's attention.

The design space of liquid software arises from issues in replicating and synchronizing the software components and their state. 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 [4]. 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 devices synchronize their state via centralized servers. Alternatively, in a decentralized approach liquid software flows directly between devices, leveraging peer-to-peer (P2P) connectivity and direct replication across devices. The granularity of the components and state that need to be migrated is another fundamental dimension.

By outlining the most important design issues and the corresponding technical choices, we provide an overview of how to build liquid software systems that can be deployed and then run across multiple devices owned by one or more users. The resulting design space also provides a useful overview of emerging technologies and frameworks that support the implementation of liquid software. More precisely, in this paper, we first provide an insight into the concept of liquid software, and connect it to emerging computing trends. Then, we explore technology alternatives and the design space needed for implementing the concept. Finally, we provide a discussion on building liquid software systems as well as related efforts.

II. Towards Liquid Software

Recent device shipment trends indicate that we will quickly move from a world in which each person has a few devices – a PC, a smartphone and possibly a tablet – to a world in which people will use dozens of connected devices in their daily lives: laptops, phones, tablets and phablets, game consoles, smart TVs, car displays, watches, augmented reality glasses, digital cameras, digital photo frames, home appliances, and so on; all of them connected to the Internet [5], [3]. Designing applications that work with such range of devices requires special considerations in their design [1]. The concept of liquid software originates from late 1990s and early 2000s [2], [6], and currently culminates in environments such as Apple Handoff [4] in which the user can, e.g., start writing an email using a smartphone, and then finish it with a laptop that has a much better keyboard for writing longer emails. We have

recently distilled the overall vision into the *Liquid Software Manifesto* [3], and described the main challenges in applying the vision to Web applications [7]. Moreover, we have also introduced an architectural style for liquid Web services [8]. All of these techniques, following the principles laid out in [9], demonstrate how liquid applications can flow from computer to computer in a simple, straightforward, and hassle-free fashion.

Reflecting the state-of-practice today to the vision above, automatic synchronization of multiple computing devices today is at best supported only partially, usually within select ecosystems only, and even then the user must 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. In such a multi-device computing environment, the users can effortlessly roam between all the computing devices that they have, with data and applications synchronized transparently between all the computing devices, potentially even spanning multiple native platform ecosystems. In particular, whenever applicable, roaming between multiple devices includes the synchronization of the full state of each application, so that the users can seamlessly continue their previous activities on any device. This requires careful consideration of the following issues:

Adaptation of the user interface to different devices and contexts. While the functions of an application may remain the same, the devices that are used for running the application may differ in terms of display capabilities and input devices. In addition, the users' ability and interest to pay constant attention will differ according to the context. Thus, the user interfaces of liquid applications should be responsive and adapt onto the set of devices where they currently run.

Synchronization of the users' persistent data. This content is commonly stored locally on each device and can be synchronized using different cloud-based storage services [10]. Unfortunately, these cloud-based storage systems are often limited to single applications, specific data types or certain native operating system ecosystems.

Synchronization of the state of the applications that move during execution or run on multiple devices simultaneously. In addition to the data consumed and produced by an application, liquid software is also concerned with the runtime (ephemeral, dynamic) state of the application itself. This runtime state includes the values of relevant variables, volatile memory storage and user interface configuration settings, which may not be necessarily persisted after the application completes, but must be migrated with the application in order to give a true sense of continuity to the user.

Partitioning. In the context of liquid applications, the balance between server- and client-side execution can change dynamically. Different devices have different capabilities, and thus optimal configurations may vary. Therefore, liquid software frameworks should offer capabilities for offloading computation from clients to servers and vice versa. Since the capabilities of computing devices may vary considerably, we anticipate both *Ultra Thin* approaches, where almost all the computation is performed on the server side, and ultra-thick designs, where the clients are completely self-contained, with no need for server components whatsoever.

Security. The user must 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 shall be able to define this in a simple, intuitive fashion. 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 can be downplayed for software running on multiple devices belonging to the same user, there is a need for assessing and evaluating to which extent existing security solutions can be applied to liquid software. An in-depth treatment of this matter is not in the scope of this paper.

TABLE I. TECHNOLOGIES AND ALTERNATIVES

		Sun Ray [11]	Joust [2]	Fluid Computing [6]	Continuity [4]	Android Baton [12]	Cloudberry [13] Cloudbrowser [14]	Continuum [15]	Liquid.js DOM [16]	Liquid.js for Polymer [17]
Architecture	**Topology**									
	Centralized	X	X			X	X		X	X
	Decentralized		X	X				X	X	X
	Layering									
	Ultra Thin Client	X					-	X		
	Thin Client									X
	Thick Client		X	X	X	X	X		X	
	Client Deployment									
	Preinstalled	X	X	X	X			X		
	On-Demand					X			X	X
	Cached						X			X
	Granularity									
	OS	X						X		
	VM/Container									
	Application		X	X	X	X	X		X	
	Component						X			X
State	**State Identification**									
	Implicit	X						X	X	
	Explicit		X	X	X	X	X		X	X
	Synchronization									
	Trickle	X	X	X			X	X	X	X
	Batch			X	X	X	X			
Liquid User Experience	**Device Usage**									
	Sequential	X		X	X	X	X		X	X
	Parallel		X	X				X		X
	UI Adaptation									
	None		X	X					X	X
	Responsive	X					X			X
	Complementary				X	X	X	X		X
	Primitives									
	Forwarding	X	X					X	X	
	Migration				X	X	X		X	X
	Forking									X
	Cloning			X			X		X	X
	Discovery									
	Shared URL		X				X	X	X	X
	QR Code									X
	Bluetooth			X	X	X				
	WiFi			X		X				
	SmartCard	X								

III. THE DESIGN SPACE OF LIQUID SOFTWARE

The liquid user experience can be implemented in a number of different ways. To sketch the design space for liquid software, we discuss relationships and dependencies between a number of design issues and alternatives (Figure 1). Moreover, Table I characterizes technology options, and positions them in the liquid software design space.

A. Topology

The *topology* of a liquid architecture can be *centralized*, with a single, well-defined host that keeps the master copy

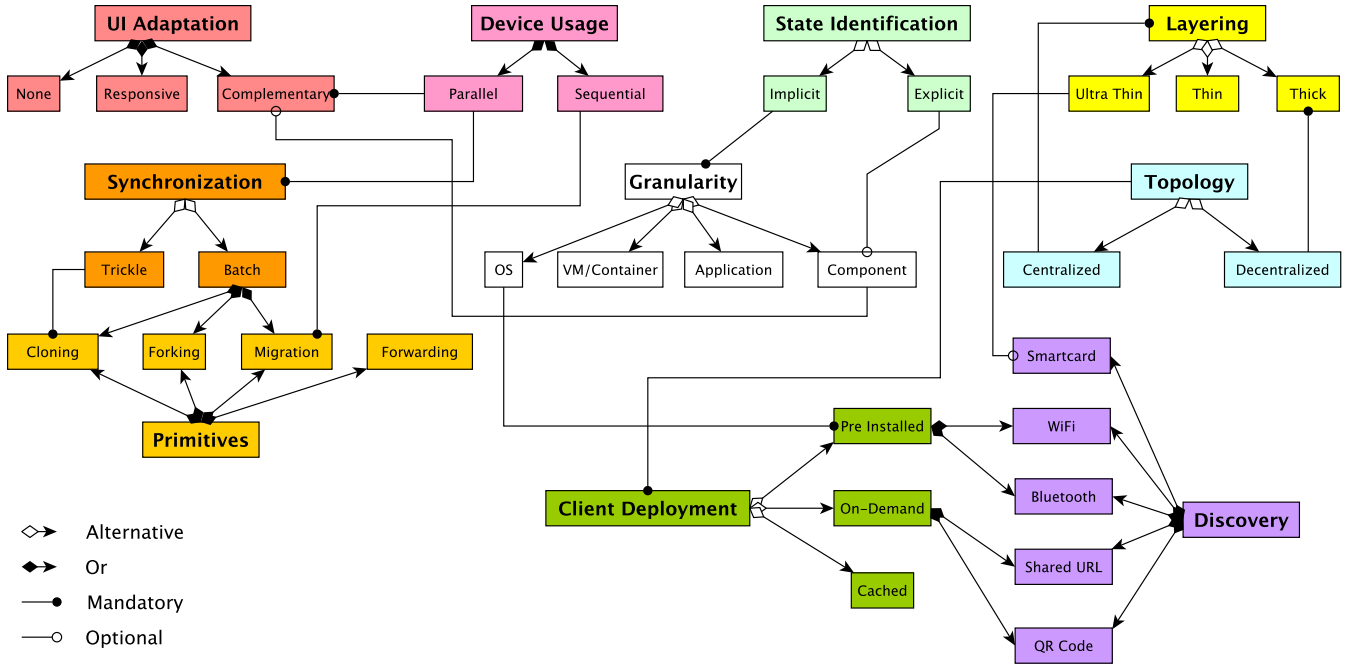

Fig. 1. Overview: the Design Space of Liquid Software. *Mandatory* arrows indicates that a child feature is required; *optional* arrows indicate that the child feature is optional; *alternative* arrows indicate that only one child feature must be selected; *or* arrows indicate that at least one child feature must be selected.

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, at the expense of the privacy of the data, which is no longer confined to user controlled devices. Liquid software thus flows up and down from the Cloud onto various user devices that are thus easy to back up and synchronize as long as a connection to the Cloud is available.

Alternatively, liquid software architectures can be designed with a *decentralized* topology, where software, applications' state 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 conflicts when synchronizing the state as there is no longer a single master copy. Likewise, while each individual user device may have perfect internet connectivity, it is unlikely that all the devices of the user are always online at the same time. Thus, special care must be taken to ensure successful migration and synchronization of state across all peered devices.

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, as only local connectivity is needed for migrating state from one device to another, and the users' 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 [10]. In real-life implementations, the borderline between the two basic topologies is not always clear. For instance, in [18], the implementation techniques forced the design to use a centralized server for communication, while conceptually the migration was designed in a distributed fashion. It should also be noted that sharing of user data, synchronization of the application state, and application deployment do not need to be organized according to the same topology, and the final architecture may be a mixture of centralization and decentralization.

The selection of the topology also depends 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 – for instance, the new target device is not online when the previously used device has been switched off. Centralized topology can introduce a store-and-forward functionality that allows sequential scenarios with a gap in between usages and devices.

Finally, an important topology-related aspect is to define how liquid software becomes aware of the set of 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, *WiFi* access point connectivity, and special purpose hardware such as *smartcards*.

B. 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.

• *Operating system level*. Implementing liquid software at operating system (OS) level is the most comprehensive but also the most complex approach; in OS level implementations, the entire operating system has been designed to support seamless process migration, state synchronization and data transfer across several computing devices running the same operating system.

• *Virtual machine/Container level*. Probably the most commonly used mechanism for migration today is to utilize virtual machines that enable moving running applications between various computing devices. The technology is widely used in data centers, e.g., to bring applications and content closer to the edges 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 the more fine-grained control on which parts of the system can be moved.

• *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 will use for implementing state synchronization.

• *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 particularly interesting when multiple devices are used at the same time. This can be an efficient way to implement the *complementary* screening scenario, where a different visual component of the same application would be deployed on a different device.

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.

C. Code Deployment

There are numerous different ways to implement client deployment 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 like Android, iOS and Windows Phone. Moreover, even Web applications in some platforms, such as Tizen [19] and Firefox OS [20] 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 browser 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. In Cloudberry [13] the applications were run by giving the URL to the Web engine, but the application code was *cached* by using the HTML5 Application Cache [21]. 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 have been commonly encountered in real-life implementations: • *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 most the functionality is on the server, and possible application updates are also server-driven. In many frameworks the client application is dynamically generated and may change as a consequence of changes on the server side. • *Thick client, on-demand deployment*. One of the main benefits of thick client applications is the support of offline operation when network connection is not available. In Web applications, this benefit can only be achieved if Application Cache is used. • *Thick client, pre-installation*. This combination resembles the traditional, solid installable binary applications. Obviously, offline use of applications is enabled. In the extreme, ultra-thin cases there is no application installation to end-user devices at all. Naturally, there need to be deployment mechanisms for the server-side applications. Conversely, in ultra-thick designs, the server might not be needed at all, since everything is managed by the client.

D. Liquid User Experience Primitives

From the user perspective the liquid software acts just like any other software, with in addition a combination of the following four primitives [9], which makes it possible to shift from a solid to a liquid user experience: • *Forwarding*: the ability of transparently forwarding the output and redirecting the input gathered on one device to the application remotely running on the other device. • *Migration*: the ability of partially or completely moving the current instance of the liquid application from a device to another effortlessly. • *Forking*: the ability of partially or completely creating a copy of the current instance of the liquid application on a different device. • *Cloning*: the ability of partially or completely creating a copy of the current instance of the liquid application on a different device (i.e., forking) *while keeping the two instances synchronized thereafter*.

E. State and Data in Liquid User Experiences

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 carried across devices. 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. 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 later [22].

No matter which approach is chosen, a procedure that synchronises 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 [10] was to allow the latest change to override any past conflicting changes in order to avoid any deadlocks or communication overhead associated with voting.

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.

User interface adaptation. The final consideration in realizing state synchronization is to determine how to render the state of the applications with user interfaces that adapt to the set of devices used to run the application [1]. Existing mechanisms and design practices such as *responsive* web design [23] pave the way to adequately treating this dimension, although still requiring careful attention and consideration from the designers and application developers. It must be kept in mind that liquid software behavior is always to some extent an illusion – a lot of technical grunt work is needed "under the hood" to maintain the users' impression that software is flowing between devices seamlessly. A significant part of the designers' and developers' work is concerned with maintaining that illusion.

F. Security Considerations

The success of computing platforms supporting liquid behavior is fundamentally dependent on security. As summarized in [3], the ability of liquid software to readily flow from device to device is both a blessing and a curse. It is a blessing because 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 users entire computing environment – most of the applications and data – being accessible from any of the user's devices can make the system vulnerable from 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, user authentication, and various other primitives that are needed for implementing security features for any liquid application. These 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 accounting.

A basic principle defined in [3] 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, Sun Ray *Ultra Thin* network terminals [11] 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 end users in different usage scenarios.

IV. Related Work

There have been numerous attempts to tackle the issues arising from multiple device ownership, with different design drivers (Table I). The term *liquid software* was coined by Hartman, Manber, Peterson and Proebsting in a technical report back in 1996 [24]. Their seminal research culminated in the design of *Joust* [2] – a system that was based on synchronizing Java applications between virtual machines running in different computers.

Fluid computing [6] denotes the replication and real-time synchronisation of application states on several devices. 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 variable 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 synchronisation. Each device has a replica of the application state, allowing the device to operate autonomously; a special synchronisation protocol is used for keeping the replicas consistent.

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. A typical example of a collaborative, multi-device, component-based, thin-client groupware system is presented in [25]. The design is based on web technologies of the time, allowing incorporation of mobile devices as well as native clients in the same system.

In the wider area of mobile computing, the authors of [26] list various trends that can be related to our work. In the advent of wearable computing, omnipresent connectivity, and increasingly smart devices, it is clear that techniques that enable seamless use of multiple devices will become fundamental.

V. 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. 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. Liquid software architectures let users take full advantage of their multiple devices. By breaking down the deployment and runtime boundaries between each device, liquid software systems allow applications and their data to move across devices seamlessly, thus making multiple device usage and ownership significantly easier than today. There are many possible alternatives to consider when designing software architecture supporting liquid software. In this paper we have presented a design space for realizing liquid software. We outlined the most important design issues and the corresponding technical alternatives. The resulting design space also provided an overview of emerging technologies and frameworks that support the implementation of liquid software.

Acknowledgments

This work has been supported by the Academy of Finland (projects 283276 and 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.

References

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

[2] J. H. Hartman, P. A. Bigot, P. G. Bridges, A. B. Montz, R. Piltz, O. Spatscheck, T. A. Proebsting, L. L. Peterson, and A. C. Bavier, "Joust: A platform for liquid software," *IEEE Computer*, vol. 32, no. 4, pp. 50–56, 1999.

[3] A. Taivalsaari, T. Mikkonen, and K. Systä, "Liquid software manifesto: The era of multiple device ownership and its implications for software architecture." in *38th IEEE Computer Software and Applications Conference (COMPSAC)*, 2014, pp. 338–343.

[4] G. Gruman, "Apple's Handoff: What works, and what doesn't," *InfoWorld*, Oct. 7, 2014.

[5] Gartner Group, "Gartner says worldwide traditional pc, tablet, ultramobile and mobile phone shipments on pace to grow 7.6 percent in 2014," http://www.gartner.com/newsroom/id/2645115.

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

[7] T. Mikkonen, K. Systä, and C. Pautasso, "Towards liquid web applications," in *Proc. of ICWE*, 2015, pp. 134–143.

[8] D. Bonetta and C. Pautasso, "An architectural style for liquid web services," in *Proc. of WICSA*, 2011, pp. 232–241.

[9] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding code mobility," *IEEE Trans. Softw. Eng.*, vol. 24, no. 5, pp. 342–361, May 1998.

[10] O. Koskimies, J. Wikman, T. Mikola, and A. Taivalsaari, "Edb: A multi-master database for liquid multi-device software," in *Proc. of the Second ACM International Conference on Mobile Software Engineering and Systems*, 2015, pp. 125–128.

[11] "Sun Ray Products," http://www.oracle.com/technetwork/server-storage/sunrayproducts/overview/index.html.

[12] K. Bell, "Baton promises to be the ultimate android app switcher," *Mashable.com*, 2014.

[13] A. Taivalsaari and K. Systä, "Cloudberry: An HTML5 cloud phone platform for mobile devices." *IEEE Software*, vol. 29, no. 4, pp. 40–45, 2012.

[14] A. Taivalsaari, T. Mikkonen, and K. Systä, "Cloud browser: enhancing the web browser with cloud sessions and downloadable user interface," in *Grid and Pervasive Computing*. Springer, 2013, pp. 224–233.

[15] "Microsoft Continuum," http://www.windowscentral.com/continuum.

[16] J.-P. Voutilainen, T. Mikkonen, and K. Systä, "Liquid.js: Middleware for liquid web applications," submitted, under review.

[17] A. Gallidabino and C. Pautasso, "Deploying stateful web components on multiple devices with liquid.js for Polymer," in *accepted at CBSE'16*.

[18] J. Kuuskeri, J. Lautamäki, and T. Mikkonen, "Peer-to-peer collaboration in the Lively Kernel," in *Proc. ACM Symposium on Applied Computing*, 2010, pp. 812–817.

[19] "Tizen Developer Pages," https://developer.tizen.org/.

[20] "FirefoxOS Developer Pages," https://developer.mozilla.org/en-US/docs/Mozilla/Firefox_OS.

[21] "w3schools.com," www.w3schools.com/html/html5_app_cache.asp.

[22] E. Brewer, "Cap twelve years later: How the" rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[23] E. Marcotte, *Responsive Web Design*. Editions Eyrolles, 2011.

[24] J. Hartman, U. Manber, L. Peterson, and T. Proebsting, "Liquid software: A new paradigm for networked systems," University of Arizona, Tech. Rep. 96-11, 1996.

[25] J. Grundy, X. Wang, and J. Hosking, "Building multi-device, component-based, thin-client groupware: issues and experiences," in *Australian Computer Science Communications*, vol. 24, no. 4, 2002, pp. 71–80.

[26] G. P. Picco, C. Julien, A. L. Murphy, M. Musolesi, and G.-C. Roman, "Software engineering for mobility: reflecting on the past, peering into the future," in *Proc. of the on Future of Software Engineering*. ACM, 2014, pp. 13–28.