Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture

Antero Taivalsaari
Nokia
Visiokatu 3, FI-33720 Tampere, Finland
antero.taivalsaari@nokia.com

Tommi Mikkonen and Kari Systä
Tampere University of Technology
Korkeakoulunkatu 1, FI-33720 Tampere, Finland
{tommi.mikkonen, kari.systa}@tut.fi

ABSTRACT
Today, the digital life of people in developed markets is dominated by PCs and smartphones. Yet, as successful as PCs and smartphones are, the dominant era of PCs and smartphones is about to come to an end. Device shipment trends indicate that the number of web-enabled devices other than PCs and smartphones will grow rapidly. In the near future, people will commonly use various types of internet-connected devices in their daily lives. Unlike today, no single device will dominate the user's digital life. The transition to a world of multiple device ownership is still rife with problems. Since devices are mostly standalone and only stay in sync in limited ways, the users will have to spend a lot of time managing them. These device management chores become much more tedious as the number of devices in a person's life increases. In this paper we look at the ongoing paradigm shift towards multiple device ownership and its implications for software architecture. We argue that the transition to multiple device ownership will eventually lead us to *liquid software* – an approach that will allow data and applications to seamlessly move between multiple devices and screens. The new era will imply significant changes in the development, deployment and use of software, opening up new opportunities in software engineering research as well.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures. C.0 [**Computer Systems Organization**]: General – *System Architectures*. D.2.12 [**Software Engineering**]: Interoperability.

General Terms
Design. Experimentation.

Keywords
Multiple device ownership; multi-device computing; multi-screen computing; liquid software.

1. INTRODUCTION
The history of computing and the software industry is shaped by disruptive periods and paradigm shifts that occur every 10-15 years or so. Back in the 1960's and 1970's, mainframe computers gave way to minicomputers. In the 1980's personal computers sparked a revolution, making computers affordable to ordinary people. In the 1990's, the emergence of the World Wide Web transformed personal computers into first class citizens on the Internet. In the end of the 1990's and in the early 2000's, the web browser became the most commonly used computer program, revolutionizing not only commerce but communication and social life as well.

In the 2000's, mobile phones became ubiquitous. Cross-platform mobile software platforms such as Java ME opened up mobile devices for third-party software development, planting the seeds for today's thriving, multi-billion mobile software industry. In late 2000's, multi-touch displays revolutionized the user interfaces of mobile devices. Highly successful software ecosystems and application businesses emerged especially around Apple's iOS and Google's Android platforms.

Today, the digital life of people in developed markets is dominated by PCs and smartphones. About 350 million new PCs and nearly one billion smartphones are sold worldwide in 2013. Yet, as successful as PCs and smartphones are, the dominant era of PCs and smartphones is about to come to an end, as we are at the cusp of yet another major disruption or paradigm shift.

2. MULTIPLE DEVICE OWNERSHIP
As summarized by Donald Norman in his 1999 book, The Invisible Computer [1], technological products have a fascinating life cycle as they progress from birth through maturity. Once a product is through the unstable days of its adolescence, the dimensions upon which the product is judged change dramatically. The same products that were highly attractive and desired in their youth can be irrelevant and ignored later in their lifecycle. In the past ten years, PCs and smartphones have evolved from high technology products to mass-market consumer commodities. While there are an almost infinite number of use cases for these devices, the fact is that they are increasingly viewed as replaceable appliances or commodities.

The end of the dominant era of PCs and smartphones. Today, the average consumer in the U.S. or Europe has two primary computing devices: a personal computer (usually a laptop) and a smartphone. Additionally, after Apple's successful launch of the iPad, many people carry a third device: a web tablet. While it may be tempting to think that today's PC and smartphone centric world will simply be extended with yet another device, in reality the number of network-connected devices that people use in their daily lives is expanding much more dramatically.

Device shipment trends indicate that number of web-enabled devices grows very rapidly. Every day, over 3.5 million new mobile devices and tablets are activated worldwide – over five times more than the number of babies born each day. We will quickly move from a world in which each person has two or three devices – a PC, smartphone and tablet – to a world in which people will use dozens of devices in their daily lives: laptops, phones, tablets and "phablets" of various sizes, game consoles, TVs, car displays, digital photo frames, home appliances, and so on – all of them connected to the Internet. These internet-

connected consumer devices can be divided broadly into six different categories, as depicted in Figure 1.

Figure 1. From two screens to six types of screens

The trend towards multiple device ownership is reflected well in a Developer Economics Q3/2013 survey in which the over six thousand developers worldwide were surveyed in order to study their platform preferences, developer attitudes, revenue models and tools of the trade [2]. According to this study, a considerable number of developers are actively looking into alternative target devices (e-readers, TVs, set-top boxes and game consoles) above and beyond desktops, smartphones and tablets in their future software development efforts. This trend will likely strengthen as other new types of gadgets and wearables such as smartwatches and intelligent eyeware become available more widely.

We believe that we are at a tipping point with connected devices, entering a new era of multiple device ownership. This new era will dramatically raise the expectations for device interoperability, implying significant changes for software architecture as well.

Multi-device ecosystems are still broken. As pointed out by Dearman and Pierce back in 2008, the transition to a world with multiple device ownership is rife with problems [3]. The way we operate a multitude of devices today is still broken. For instance, the need to synchronize and back up data (e.g., photos, music and documents) between multiple devices is a major hassle. Likewise, the requirement to manually define e-mail accounts, web bookmarks, RSS feeds, and other personal preferences and settings for each individual device can be painful. The need to explicitly install and then later frequently upgrade the necessary applications for all the devices separately can also be time-consuming. Furthermore, incompatibilities between devices abound; applications intended for one mobile platform do not run on other platforms; applications bought from one app store cannot be installed in other types of devices. These problems get worse and worse almost exponentially as the number of devices in a person's daily life grows.

Systems such as Apple's *iCloud* (http://www.icloud.com) and *Google Sync* (http://www.google.com/sync) are 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 "liquid" usage of software is not generally supported yet, although such user interface concepts have been presented in a number of forums before [4, 5].

Many of the hassles and limitations associated with multi-device usage arise from the broad variety of devices with different screen sizes, input mechanisms and usage patterns. With screens ranging from tiny wristwatch or wristband displays to 4K Ultra-HD TV screens, and input mechanisms ranging from T9 keypads and remote controls to touch displays and conventional QWERTY keyboards, one user interface solution simply does not fit all. This makes it difficult to migrate live applications meaningfully from one device to another. The basic technical challenges in this area have been summarized well by Thevenin and Coutaz [6]. Although design approaches such as responsive web design [7] make it easier to create software that automatically adapts to different types of target devices, the fact is that applications intended for a specific type of a device do not necessarily make sense at all on other, entirely different types of devices.

3. LIQUID SOFTWARE

We believe that multiple device ownership should be as casual, fluid and hassle-free as possible. By *casual computing*, we refer to a model of computation that is constantly available, capable of delivering meaningful value even in a few moments, without requiring active attention from the user's part [8]. Casual computing is as much a mindset as a specific set of technologies – the idea is to make technologies themselves so natural, effortless and calm that they will effectively disappear.

A central aspect of a true casual computing experience is the ability to move fluidly from one device to another. By *liquid software*, we refer to an approach in which applications and data can flow from one device or screen to another seamlessly, allowing the users to roam freely from one device to another, no longer worrying about device management, not having their favorite applications or data, or having to remember complex steps. In a true liquid software environment, device management chores such as backups, application installation, application upgrades, restarting the recently used applications, account migration, copying settings across devices, or other similar activities that burden the daily lives of users today should be things of the past.

Figure 2. Liquid software illustrated

Liquid software usage is illustrated in Figure 2. This scenario is taken from a demo video by Corning [9]. The video was created primarily as a showcase for future glass technologies; however, many of the software usage scenarios in it illustrate the concept of liquid software well. In the figure, the girl is interactively transferring and synchronizing the music player app and its theme from her tablet to the car's audio system. The same video has a

number of other compelling examples of liquid software usage, e.g., in a school environment and in doctor's office.

There are numerous use cases and examples of liquid software. In the broadest scenario, it is possible to envision an entire operating system to be fully liquid ("Liquid OS"), meaning that all the user sessions, including the state of every open application can be transferred live from one computer to another, or potentially used from multiple computing devices simultaneously; likewise, all the user's files and other data would be equally accessible on all devices. Given the rapidly growing number of computing devices and the management hassles associated with multi-device usage, we envision that this is the way personal computing operating systems will have to behave in the future.

4. LIQUID SOFTWARE MANIFESTO

The "Liquid Software Manifesto" below summarizes our key requirements for a seamless multi-device ecosystem:

1) In a truly liquid multi-device computing environment, the users shall be able to effortlessly roam between all the computing devices that they have.
2) Roaming between multiple devices shall be as casual, fluid and hassle-free as possible; all the aspects related to device maintenance and device management shall be minimized or hidden from the users.
3) The user's applications and data shall be synchronized transparently between all the computing devices that the user has, insofar as the application and data make sense for each device.
4) Whenever applicable, roaming between multiple devices shall include the transportation / synchronization of the full state of each application, so that the users can seamlessly continue their previous activities on any device.
5) Roaming between multiple devices shall not be limited to devices from a single vendor ecosystem only; ideally, any device from any vendor should be able to run liquid software, assuming the device has a large enough screen, suitable input mechanisms, and adequate computing power, connectivity mechanisms and storage capacity.
6) The user shall remain in full control regarding the liquidity of applications and data. If the user wishes certain functionality or data to be accessible only on a single device, the user shall be able to define this in a simple, intuitive fashion.

Today, automatic synchronization of multiple computing devices is still supported only partially and is something that the user needs to turn on explicitly. We believe that by the end of this decade, multi-device usage will be so ubiquitous that automatic synchronization will be the norm rather than the exception.

5. BACKGROUND AND RELATED WORK

The term *liquid software* was coined by Hartman, Manber, Peterson and Proebsting in a technical report back in 1996 [10]. Their focus was primarily on enabling the flexible use of network-transported code on top of the Java platform [11]. The use cases mentioned in their documents include, e.g., the ability to simplify remote execution since it gives one site the ability download the modules it needs to access the resources at another site. Remote execution, in turn, would make it possible to do software installation, diagnostics, and maintenance at a distance. As another example, liquid software was to facilitate seamless updates, thereby supporting just-in-time downloading of software updates, as well as the evolution of software over time.

Sun's (now Oracle's) Sun Ray platform [12] is a good example of a "semi-liquid" computing environment in which the users can easily transport their entire network computing sessions from one physical Sun Ray terminal to any other by inserting their JavaCard™ into the terminal's smart card reader. Sun Ray architecture is based on "screencasting", i.e., transmitting live, compressed screen images from the servers to the Sun Ray terminals; no computation or data (other than screen images) are actually transferred away from the servers. As a result, Sun Ray terminals are very secure but utterly dependent on a network connection; offline use of Sun Ray terminals in the absence of an active network connection is not possible.

More recently, a number of systems have emerged specifically to facilitate dynamic synchronization of data between different types of computing devices. Amazon Cloud Drive, Apple iCloud, Dropbox, Google Sync and Microsoft Skydrive all share the same basic architecture, allowing the users to store files in the cloud and share those files easily between different devices. In addition, many of these systems make it possible to synchronize device settings (bookmarks, account settings, contacts, calendar reminders, etc.) as well. Apple's iCloud even makes it possible to synchronize the set of installed applications across multiple iOS devices.

From the user experience perspective, liquid software can be viewed as a form of Computer-Supported Collaborative Work (CSCW) – a research area that focuses on how collaborative activities and their coordination can be supported by means of computer systems. CSCW is an old research topic in Human Computer Interaction area that dates back to the mid-1980's [13]. However, an important difference between CSCW and liquid software is that in liquid software multi device usage is related primarily to *devices owned and used by a single person*, whereas in CSCW the focus is on collaborative device usage between *multiple users* ("groupware").

6. IMPLICATIONS FOR SOFTWARE ARCHITECTURE

Liquid software may still seem like science fiction. However, the basic technical ingredients and enablers for supporting liquid software are already in place. At the implementation level, liquid software is all about *virtualization*, i.e., detaching the software and data from their physical manifestations, and making it possible to use them on a variety of underlying platforms and devices.

Technology enablers. Liquid software relies extensively on code and data that are provisioned and synchronized dynamically over the network. Cloud computing, wireless networks and local connectivity technologies such as Bluetooth LE or Wi-Fi Direct make it possible to perform provisioning and synchronization over the air to a multiplicity of devices, as well as optionally enable dynamic, over-the-air synchronization of devices directly via local connectivity.

More recently, Backend as a Service (BaaS) systems such as Parse (http://www.parse.com/) and Database as a Service systems such as Firebase (http://www.firebase.com/) have emerged to make cloud-based data storage and push notifications readily

available, raising the abstraction level and considerably lowering the bar for implementing dynamic synchronization of devices.

At a more elementary level, portable, retargetable code, fast interpretation and adaptive just-in-time compilation techniques form the technological foundations that make it feasible to run the same software in different devices built on different hardware, different underlying operating systems and different application frameworks. Likewise, web technologies and HTML5 play a central role in facilitating the development of a cross-platform multi-device ecosystem with transparent synchronization across native platforms. While in the near term multi-device ecosystems will likely be built on top of existing closed ecosystems (e.g., Apple's iOS platform), a true multi-device ecosystem should not be limited to devices from a single vendor only. The user interface challenges related to multi-device usage, e.g., the difficulties in making the same software able to run fluidly on a variety of devices with different screen sizes and input modalities can be addressed partially via responsive web design [7].

Basic architectural dimensions and options. The basic architectural options for realizing the liquid software vision can be summarized as follows:

1) *Master-slave vs. multi-master architecture*. One of the most fundamental dimensions in designing the underlying architecture for liquid software is whether data synchronization and code provisioning in the system are based on a master-slave or multi-master architecture.

   In *master-slave architecture*, code provisioning and data synchronization are performed via a shared server (usually) in the cloud. This shared server holds the master copies of code and data, and pushes changes made in one device (slave) to the other devices. This is the most common architecture approach for device synchronization today.

   In a *multi-master architecture*, code provisioning and data synchronization are performed on a peer-to-peer (P2P) basis, with no single device computer dominating the overall architecture. These types of device synchronization systems are still relative rare because of limited device support for local connectivity technologies and the challenges associated with multi-master synchronization in the presence of potentially intermittent or unreliable network connections.

   Note that hybrid approaches are possible as well. For instance, it is possible to envision a system in which code is provisioned from a centralized server, while data synchronization is performed on a peer-to-peer basis whenever local connectivity mechanisms are available.

2) *Native vs. web-based ecosystem architecture.* Another important dimension in designing a liquid software environment is whether the system is built for a specific native ecosystem only, or whether it is intended to be a "platform of platforms", i.e., capable of running liquid applications on a number of different native platforms. In practice, the latter type of an approach would most likely be built on the web-based (HTML5) ecosystem, since HTML5 it is only a common denominator platform that is supported widely enough across the industry. As summarized in [2] and in our earlier papers [14, 15, 16], the feature set offered by HTML5 is still rather limited for using the Web as a full-fledged software platform. In contrast, performance issues associated with HTML5 are much less of an issue, since the performance of JavaScript virtual machines has increased dramatically over the past five years.

3) *Centralized vs. distributed computing and rendering and architecture*. Technically, the simplest approach for implementing a multi-device software platform would be to use a centralized computing and rendering architecture and then rely on screencasting from the centralized server to all the client devices, in the same fashion as in the Sun Ray system mentioned earlier [12]. In such a system, no code or data would have to be provisioned away from the centralized server(s). In practice, such architecture would be wholly impractical for today's mobile devices that do not have inexpensive, 100% reliable network connections available.

Note that in addition to the basic dimensions summarized above, there are many other architectural considerations, e.g., related to connectivity and authentication choices and tradeoffs.

Security implications. 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. In general, privacy/security/authentication issues in a liquid software environment can be even more challenging than in today's cloud computing systems.

Traditionally, the outside world can only interact with a computer through well-defined interfaces presented by the fixed set of programs installed on the computer. These restrictions allow the computer to protect itself from external attacks. Liquid software eliminates this protection, since the code that a computing device runs is provisioned dynamically. Without proper precautions the computer may import and run unsafe code, thereby opening up the computer to abuse through misuse of its resources, e.g., the code may consume excessive amounts of memory or CPU, access memory or files that it should not, transmit confidential information to the outside world, and so on.

Furthermore, the very 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 the privacy perspective. For instance, if even one of the user's devices is stolen, there is a possibility that his entire computing environment will be compromised.

The solutions to the problem of unsafe liquid software fall into three categories: (1) user control, (2) implicit trust, and (3) verified access. User control is the simplest of the three to implement, and is the one currently used in mobile code systems such as Java. In this solution the user controls the resources that the liquid software may access, either by setting up access control lists for the resources, or by responding to dialog boxes that describe the type of resource that the liquid software requires and allow the user to decide whether or not to grant the request. As obvious, such solutions must not be made too intrusive from the user's viewpoint, since this would go against the key value proposition of effortless, calm and casual multi-device use.

Architectural examples and explorations. In our own earlier work, we have explored and experimented with many of the architectural options [17, 18, 19]. None of these systems is a full-

fledged liquid software system yet. However, all of these systems exhibit many of the qualities that a system fulfilling our liquid software manifesto should have.

Cloudberry is a proof-of-concept HTML5 mobile device platform – developed at Nokia in 2009-2011 – in which all of the device's user functionality is downloaded and cached dynamically from the Web [17]. The dynamically downloaded functionality includes all the applications and even the entire top-level user interface. The system was built specifically to support multiple device ownership with the mindset that all the data and most of the application functionality are accessible from any HTML5-enabled device or even from a generic desktop web browser. Dynamic synchronization of application state is supported via database architecture with built-in push notifications. Master-slave architecture is used, meaning that master copies of applications and all the data reside in the cloud. Offline caching is used for enabling offline use of applications when network access is not available. Some examples of liquid applications were built by explicitly using the synchronization capabilities provided by the underlying database solution.

An example of a peer-to-peer multi-device environment was created earlier by extending the world-and-wormholes metaphors of the *Lively Kernel* [20, 21]. Lively Kernel applications live in an environment referred to as the *world*. A world is a visual container that can simultaneously host various applications and objects. Different worlds are connected with wormholes. Objects and applications can be transported from one world to another via those wormholes. While the original Lively Kernel implementation was restricted to worlds that are hosted in the same computer, the concept was later extended to a distributed version in which wormholes can link Lively Kernel worlds located in different computers [18]. At the implementation level even this version utilized a shared server instead of P2P access in order to bypass the same origin policy security restrictions (http://www.w3.org/Security/wiki/Same_Origin_Policy) of the web browser. Nowadays, it would be possible to implement a purely P2P based multi-device environment utilizing WebRTC (http://www.webrtc.org/).

A design based on a liquid software framework using mobile HTML5 agents was introduced in [19]. In this implementation, web applications can store their internal state in the server for future use, and their executable code can be relocated to a different computer with the internal state of the application. The applications can also continue their execution while being stored in the server, and the running applications can later be retrieved back to a browser. Although our current applications do not include autonomous migration, the proposed approach supports it at the conceptual level. In our current implementation, a single agent instance moves from host to host, but it would be easy to change the behavior so that a new copy is created when needed; for instance, when a browser fetches the agent from the agent server, a new copy could be created and the original application could also continue its execution in the agent server. Since agents can run both with and without UI, the architecture has to be designed to keep user interface separate from execution. HTML5 provides a good foundation for this separation. Similarly to most HTML5 applications, our HTML5 agents are composed of two major parts, 1) declarative description of the user interface in a form of HTML, CSS and image files, and 2) JavaScript files describing the executable code [19].

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. 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 than today. Unlike today, no single device will dominate the user's digital life.

The transition to a world with multiple device ownership is still rife with problems. When using multiple devices, the average user today will still have to spend a considerable amount of time copying and backing up data, installing and upgrading apps, setting up e-mail and social media accounts on each device, relaunching applications to resume earlier work, and performing various other tedious device management chores. The time spent on these chores can increase almost exponentially as the number of network-connected devices in a person's life increases.

The trend towards multi-device ecosystems is already visible in the increasing popularity of systems such as Apple's iCloud, Google Sync and Microsoft Skydrive. While these systems focus on making it possible to synchronize the user's data and settings across devices belonging to the same native ecosystem, we believe that multi-device synchronization should not be limited to files or only to devices belonging to a single native ecosystem.

In this paper, we presented our *liquid software manifesto*. By liquid software, we refer to a multi-device software experience that can seamlessly and effortlessly "flow" from one device to another. Liquid software entails a virtualized but personal computing experience that is independent of any particular device or OS platform, allowing the users to seamlessly roam and continue their activities on any available device or computer.

Although liquid software may still seem like science fiction, the technical ingredients and enablers for realizing the vision are already in place. A number of architectural choices were discussed in this paper to highlight the basic technical options and dimensions. Moreover, we summarized a number of proof-of-concept systems that demonstrate the forthcoming changes in the development, deployment and use of multi-device software.

In summary, we believe that by the end of this decade, seamless multi-device operation will be the norm rather than an exception. Paraphrasing Mark Weiser [7], ultimately multi-device usage will become so seamless and ubiquitous that "it will weave itself into the fabric of everyday life until it is indistinguishable from it". This is simply how computing devices should work from now on. We hope that this paper, for its part, encourages people to continue work in this exciting area.

8. REFERENCES

[1] D. A. Norman, *The Invisible Computer: Why Good Products Can Fail, the Personal Computer Is So Complex, and Information Appliances Are the Solution*. The MIT Press, 1999.

[2] VisionMobile, Ltd., Developer Economics Q3 2013, State of the Developer Nation, July 2013. URL: http://www.visionmobile.com/product/developer-economics-q3-2013-state-of-the-developer-nation/.

[3] D. Dearman and J.S. Pierce ,"It's on my other computer!": computing with multiple devices. Proc. *CHI'2008* (Florence, Italy, April 5-10), 2008, pp. 767-776.

[4] M.A. Nacenta, D. Aliakseyeu, S. Subramanian, and C. Gutwin, A comparison of techniques for multi-display reaching. Proc. *CHI'2005* (Portland, Oregon, USA, April 2-7), 2005, pp. 371-380.

[5] S. K. Kane et al., Exploring cross-device web use on PCs and mobile devices. Proc. *Interact 2009*, pp. 722–735.

[6] D. Thevenin and J. Coutaz, Plasticity of user interfaces: framework and research agenda. In *M.A. Sasse, C. Johnson (eds), Human-Computer Interaction – Interact'99*, IOS Press, 1999, pp. 110-117.

[7] E. Marcotte, *Responsive Web Design*. A Book Apart, 2011.

[8] M. Weiser, The computer for the 21st century. *Scientific American*, September 1991, pp. 94-104.

[9] Corning, Inc., A Day Made of Glass 2, 2012. URL: http://www.youtube.com/watch?v=jZkHpNnXLB0.

[10] J. J. Hartman, U. Manber, L. L. Peterson, and T. A. Proebsting, Liquid software: a new paradigm for networked systems. *Univ. of Arizona Tech Report* TR 96-11, 1996.

[11] J. J. Hartman, P. A. Bigot, P. Bridges, B. Montz, R. Piltz, O. Spatscheck, T. A. Proebsting, L. L. Peterson, and A. Bavier, Joust: a platform for liquid software. *IEEE Computer*, April 1999, pp. 50-56.

[12] Oracle, Inc, Sun Ray Clients and Oracle Desktop Virtualization Clients. URL: http://www.oracle.com/us/technologies/virtualization/sun-ray/overview/index.html

[13] J. Grudin, Computer-supported cooperative work: history and focus, *IEEE Computer*, May 1994 pp. 19–26.

[14] A. Taivalsaari and T. Mikkonen, The web as an application platform: the saga continues. Proc. *Euromicro Conference on Software Engineering and Applications* (Oulu, Finland, August 30 – September 2), 2011, IEEE Computer Society, pp. 170-174.

[15] A. Taivalsaari, T. Mikkonen, M. Anttonen, and A. Salminen, The death of binary software: end user software moves to the web. Proc. *9th International Conference on Creating, Connecting and Collaborating through Computing* (C5'2011, Kyoto, Japan, January 18-20), 2011, IEEE Computer Society, pp.17-23.

[16] M. Anttonen, A. Salminen, T. Mikkonen, and A. Taivalsaari, Transforming the web into a real application platform: New technologies, emerging trends, and missing pieces. Proc. *26th ACM Symposium on Applied Computing* (SAC'2011, TaiChung, Taiwan, March 21-25), 2011, ACM Press, proceedings vol 1, pp.800-807.

[17] A. Taivalsaari and K. Systä, Cloudberry: HTML5 cloud phone platform for mobile devices. *IEEE Software*, July/August 2012, pp.30-35.

[18] J. Kuuskeri, J. Lautamäki, and T. Mikkonen, Peer-to-peer collaboration in the Lively Kernel. Proc. *25th ACM Symposium on Applied Computing*, 2010, pp. 812-817.

[19] K. Systä, L. Järvenpää, and T. Mikkonen, HTML5 agents – mobile agents for the web. Proc. *International Conference on Web Information Systems and Technologies 2013* (WebIST'13, Aachen, Germany, May 8-10), 2013, pp. 37-44.

[20] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen, The Lively Kernel – a self-supporting system on a web page. Proc. *Workshop on Self-Sustaining Systems* (S3'2008, Potsdam, Germany, May 15-16, 2008), LNCS5146, Springer-Verlag, 2008, pp. 31-50.

[21] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz, Web browser as an application platform: The Lively Kernel experience. *Sun Microsystems Laboratories Tech Report* TR-2008-175, 2008.