Pers Ubiquit Comput (2014) 18:481–488
DOI 10.1007/s00779-013-0666-y

ORIGINAL ARTICLE

Device-Independent Architecture for ubiquitous applications

Jacek Chmielewski

Received: 15 November 2012 / Accepted: 20 March 2013 / Published online: 23 April 2013

Abstract The part of the Internet of Things composed of devices that directly interact with users has grown considerably in the past years. With new smartphones, tablets and other Internet-enabled devices that appear on the market, this trend is still increasing. However, existing application development processes and tools, designed for single device applications, do not allow developers to fully and efficiently address this opportunity. Applications are developed for a particular type of devices or a particular programming platform. This limits the number of potential users and makes it difficult to seamlessly use an application on multiple devices owned by users. To take full advantage of the Internet of Things, applications should be able to run on any device—they should be ubiquitous. In this paper, we present a concept of *Device-Independent Architecture*, which provides separation of applications from devices and facilitates development of device-independent applications. Additionally, the separation introduced by the Device-Independent Architecture enables implementation of multi-device scenarios where a single application employs multiple devices at the same time. The experiment described in the paper proves that such device-independent applications indeed may be used on any suitable device—they have a chance to become ubiquitous.

Keywords Device independence · Multi-device applications · User interface adaptation · Context-aware applications · Internet of Things · Internet of Services

J. Chmielewski (✉)
Department of Information Technology, Poznan University of Economics, Poznan, Poland
e-mail: chmielewski@kti.ue.poznan.pl

1 Introduction

The term *Internet of Things* (IoT) is usually used to describe systems composed of multiple sensors and actuators. According to Marc Weiser's [1, 2] 'invisible servant' rule, these devices should operate in the invisible (calm) way influencing the real world, but not interacting directly with people. People only notice the results of device activities rather than the devices themselves. However, this is not the only side of the IoT. According to the definition provided by EC [3], the IoT also includes all the devices used directly by people: smartphones, tablets, smart TV, intelligent home appliances, public interactive touch panels, etc; essentially, any electronic device connected to the Internet and used to interact with a user. Such devices may provide a number of sensors, but the main differentiator is that they provide user interaction channels (UICs) that allow interacting directly with a user, not only with the environment surrounding the user. In this paper, we focus on this side of the IoT, and to differentiate from sensors and actuators, we call these devices *end-devices*. End-devices are used by users to access applications that expose their functions using user interfaces (e.g., visual representation of buttons on a touch screen or vibration as a tactile feedback). In this paper, users of these applications are called *end-users*, and consequently the applications are called *end-user applications*.

The growing number of software platforms and increasing diversity of end-devices [4, 5] make the development of end-user applications a difficult and time-consuming task. Developers can either target only a few most popular platforms excluding some end-users and limiting the reach of their end-user applications or develop a large set of separate end-user applications targeting different platforms and end-device classes. Despite being

economically hard to justify, the provision of application functionality to end-users using a wide range of end-devices will become even more difficult as new software platforms emerge and gain acceptance (Windows Phone, bada, Firefox OS, Tizen) and new classes of end-devices connect to the Internet—not only smartphones and tablets, but also car infotainment systems (BMW ConnectedDrive), smart TVs (Samsung SmartTV), intelligent home appliances (LG Smart ThinQ series), interactive coffee tables (MS PixelSense), etc. In consequence, there is a need for a more universal approach to the implementation of end-user applications—an approach that will lower the development effort and will make end-user applications easily accessible to end-users despite the diversity of end-devices. One way to make it possible is to build applications that are independent of end-devices. With such a device-independent approach, developers need to implement only a single version of an application and end-users are not tied to a specific device—they may use their applications on any suitable end-device. Essentially, the device independence would help make end-user applications ubiquitous.

The remainder of this paper is organized as follows. Section 2 contains some background information on how device independence is addressed in today's solutions. In Sect. 3, the Device-Independent Architecture (DIA) is introduced and described. Section 4 contains a description of an experiment verifying the feasibility of the Device-Independent Architecture. Section 5 concludes the paper and provides an overview of future research directions.

2 Device independence

Device independence means that functions of an end-user application are available on any suitable end-device without the need to modify the application itself. Commonly, the device independence is achieved by separating the application from lower layers of the device.

Separation from the hardware can be provided by an operating system (OS) [6]. An OS abstracts device hardware features using device drivers and delivering hardware-related functions in a form of an API. Applications using the API are hardware independent, but become OS dependent. There is large number of different OSs available on the market, and each OS has its own API and its own limitations to software platforms that can be used by developers. Therefore, the introduction of the OS as a separation layer does not provide true device independence—it does not allow building ubiquitous end-user applications.

Separation from the diversity of OSs can be provided by an additional layer—*universal runtime environment* (URE) such as Java Virtual Machine, Flash engine or a Web browser. UREs are available on multiple different OSs and are characterized by the fact that they wrap the OS API with their own API. Also, UREs usually limit development options to a specific programming language (e.g., Java, ActionScript, HTML5/JavaScript). Applications developed for a particular URE are hardware and OS independent, but become dependent on the availability of the URE. With the increasing number of mobile devices such as smartphones and tablets [4] and declining support for many once-popular universal runtime environments [7], the Web emerges as a leading platform for device-independent applications [8, 9]. Therefore, most of the current research on device-independent applications is focused on various aspects of Web applications: from differences among Web browsers [10], to new HTML5 APIs, CSS3 properties and cross-browser Java Script libraries that try to unify [11] and enhance Web browser behavior [12], to new server-side frameworks [13].

However, the main problem with the Web-based approach is that it requires powerful end-devices capable of supporting all separation layers, Web runtime and additional JavaScript libraries. The resource constraints are especially important for embedded and mobile devices. Embedded devices usually have limited processing capabilities, which makes them a hard target for Web applications (cf. webinos efforts to put their runtime on Arduino-based devices [12]). On the other hand, mobile devices may have powerful processors, but they depend on battery life which can be significantly limited by increased processing requirements of Web applications. This is one of the reasons why Apple mobile devices never supported Flash technology [14].

Achieving the device independence of end-user applications by introduction of additional separation layers seems to be a dead-end—especially for the billions of mobile and embedded IoT end-devices. To get past those limitations, we propose a new architecture for device-independent applications, which ensures application-device separation also in the field of constrained device resources.

3 Device-Independent Architecture

To make it possible to build device-independent, ubiquitous end-user applications, we propose to approach the problem in a different manner. Since applications are constrained by limited capabilities of end-devices (processing power, battery life, etc.), we see the solution in running applications outside of the device.

Additionally, to avoid adding new separation layers that increase the complexity of the solution, we want to maintain the link between an application and a device via a generic protocol—reusing proven strategies from the

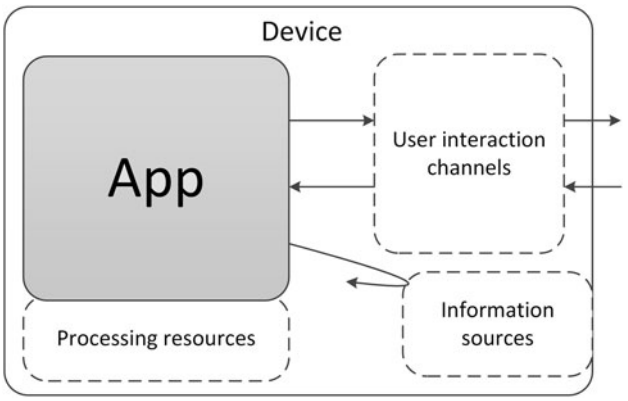

Fig. 1 Device features used by application running on a device

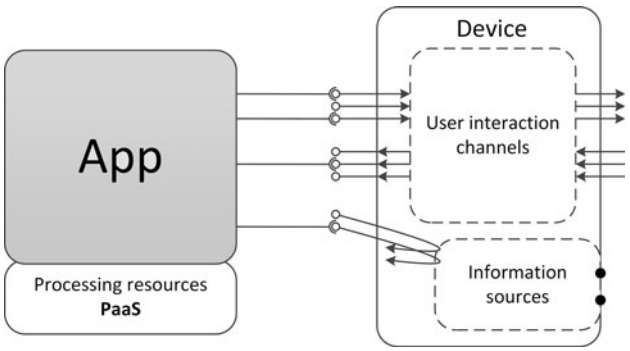

Fig. 2 Device-Independent Architecture

computer network domain where well-defined protocols enabled multiple heterogeneous network nodes to cooperate and form the Internet [15]. This is the basis of the DIA.

3.1 Running outside of the device

Application running on a device uses a number of features provided by the device (see Fig. 1). All these features can be assigned to one of three groups: (1) *processing resources*—CPU, memory, storage; (2) *information sources*—device sensors providing data such as location, temperature, light intensity; (3) *UICs* (input and output)—screen, speakers, vibration, keyboard, mouse or another pointing solution, touchscreen, orientation sensors (e.g., for gestures), microphone, camera, etc.

To make it possible to run an application outside of a device, we have to provide replacement for these three classes of device features. The idea behind the DIA originates from Service-Oriented Architecture [16], where software systems are decomposed into atomic services, while processes use necessary services without knowing implementation details of systems that provide these services. Similar approach can be used to functionally decompose devices and provide their features to applications as services. To maintain device independence of applications, applications should not rely on how these services are implemented. The only thing that should be known to applications is how to use these services.

Processing resources can be provided as cloud-based services in the form of a Platform as a Service [17] (PaaS). To maintain device independence of applications it is not necessary to specify a protocol for accessing a PaaS infrastructure. Each PaaS implementation has its own requirements and API, but the selection of a particular PaaS will only influence the choice of a programming language and development patterns. It will not hinder the device independence of an application developed on the selected platform. Therefore, it is up to the developer to decide which PaaS will be used to develop and run his applications.

Information sources cannot exist without a device, but can be exposed as services accessible with a simple REST GET method (for one time access) or a protocol based on a subscriber/provider pattern [18] (for continuous access). Such generic protocol ensures proper application-device separation while maintaining the ability to use data provided by device sensors. The types of data provided by information sources can be described using syntactic and semantic service description languages such as WSDL or USDL [19].

User interaction channels can be divided into output interaction channels provided to applications as data sinks accepting specific data formats (e.g., visual data for screen, audio stream for speakers, etc.) and input interaction channels exposed as services accessible according to a protocol based on the observer pattern [20]. Again, types of interaction events provided by these services can be described using appropriate service description languages.

The overall diagram of the DIA is presented in Fig. 2. It maintains all components necessary to run an application presented in Fig. 1. In traditional approaches, the application device separation was hindered by different APIs provided by different OSs and UREs. The decomposition into atomic services makes it easier to define self-contained access protocols that could be standardized and natively implemented in end-devices or added to end-devices in a form of a universal native application (*device-independency driver*).

One could argue that the proposed DIA is merely an implementation of the cloud computing concept already adopted by initiatives such as Google's Chrome OS or Mozilla's Boot to Gecko (now Firefox OS). But the key point of the DIA is not in just running an application in the cloud. Almost all Web applications follow this approach. The key point of the DIA is to enable an application running in the cloud to use device-specific features of an end-

device without making any compromises on the device independence of the application.

3.2 Multi-device capabilities

Provision of device features in a form of services has additional advantages. From the point of view of an application, an end-device is seen not as a monolithic device, but rather as a set of services. Such a set may include services provided by multiple end-devices. This fact opens a wide range of scenarios in which a single application becomes a multi-device application:

- *Complementary services scenario* An application interacts with a user using a single device, but employs information sources from other devices. For example, a photo capture application could provide its UI on a digital camera, but retrieve user location information from a smartphone.
- *Redundant services scenario* With similar services provided by multiple devices an application may choose services providing the best data (e.g., the most accurate location) or interaction channels best suited for interaction with a particular user in a given usage context (e.g., display information on a car dashboard instead of a smartphone screen while the user is driving).
- *Multi-device UI scenario* If multiple devices provide output and input UICs, an application may distribute fragments of its UI to separate devices choosing the best composition of available interaction channels. For example, an electronic program guide application could present visual information on a TV, but interact with a user using input interaction channels on a smartphone (e.g., gestures detected by smartphone g-sensor or gyroscope).

Implementation of multi-device scenarios require that applications know what end-devices (and what services provided by these devices) are available at a given moment. Therefore, the DIA includes a register of end-devices and a corresponding service that can be queried by applications [21].

3.3 Interaction with the user

The device independence brings multiple benefits. However, to provide application functionality in a usable manner, a user interface of an application has to be adapted to capabilities of a set of available output and input user interaction channels. There are two general approaches to this problem: (1) An application provides only an abstract model-based UI description that is used at runtime by a generic UI generation service to generate a final UI for a particular interaction channel [13, 22–24], and (2) generation of a final UI is done directly by an application supported by services that provide information about capabilities of targeted user interaction channel. The first approach is suitable for typical CRUD applications, but it is difficult to implement for complex interactive systems [25]. The second approach is best suited for custom and complex UI (e.g., games). In the second approach, an end-device does not have to interpret the UI and is responsible only for presenting the final UI (e.g., for complex visual UI, it can have a form of an image or a video stream). This approach is already used by OnLive [26, 27] and Gaikai [28] cloud gaming platforms.

4 Feasibility experiment

The goal of this experiment is to find out whether the presented approach to device independence allows maintaining functionality and usability of a native mobile application. Consider a crowdsourcing [29] application which exploits gamification ideas [30]. The application is implemented as a game where a player gains credibility points by performing quests. The quests are designed to gather real-life data at specific locations. Therefore, a player has to travel to a specific location and provide requested information (a photo or a sound recording). The application is implemented as a native Windows Phone application and is available only for smartphones with the Microsoft mobile OS. Such application may gain a lot by becoming ubiquitous—i.e., independent of end-devices used by users. Therefore, it is a great test case for the Device-Independent Architecture.

The application prototype used in this experiment uses a static UI composed of three main screens (see Fig. 3). The first screen presents a credibility ranking (Credibility Stack) and provides a link (See quests…) to available quests (Quests list). Quests list screen presents a list of available quests and a list of quests completed by the user. Selecting a quest opens the third screen (Quest details) with detailed description of the quest and quest location. The second variant of the Quest details screen (rightmost image in Fig. 3) is presented only if the user is at the required location. It provides an interface for submitting the requested data.

4.1 Device-independent prototype

With DIA, the selection of a backend technology used to implement the application is not limited. The native application is implemented in MS C# and is using MS WCF Data Services running on a Windows Server 2008. It was decided to reuse most of the code and retain the

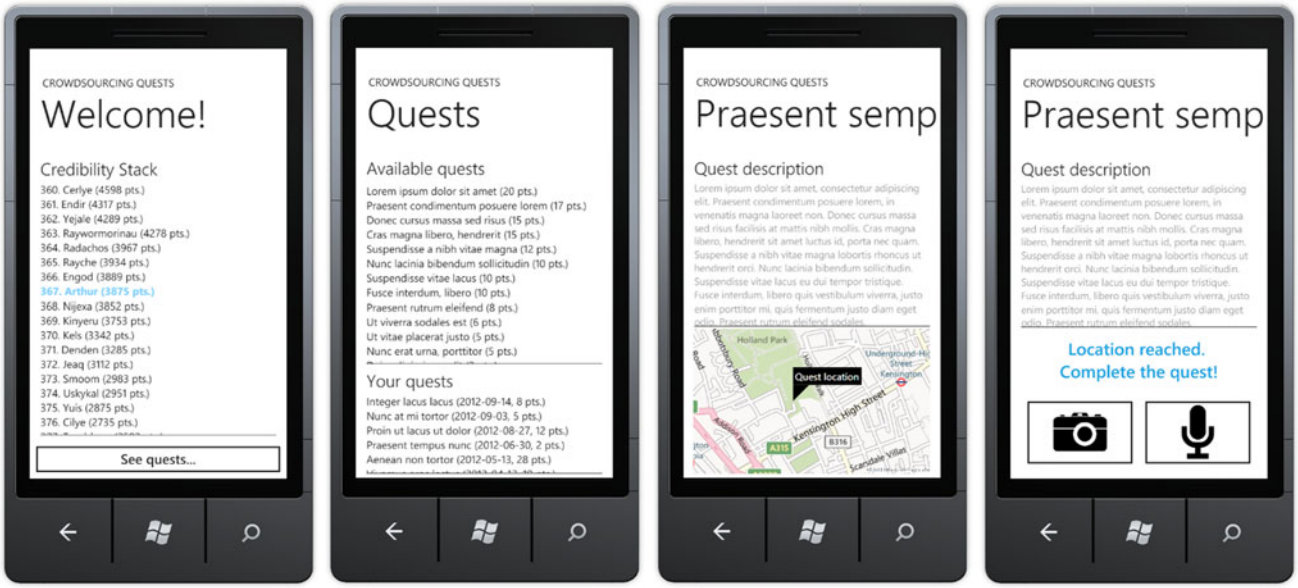

Fig. 3 User interface of the crowdsourcing application

backend infrastructure. The application code was moved to the server and modified to use HTTP protocol to communicate with services exposed on end-devices. Services on end-devices were implemented within a universal device-independency driver (native) and may be accessed via HTTP using the REST approach.

The API calls used by the native application to access geolocation information, device touchscreen, camera and microphone were replaced by calls to equivalent services. In the case of the geolocation function, the application needs the information only to check whether a user is allowed to complete a quest (i.e., a user is at the required location). Therefore, it was enough to implement a REST GET method for accessing this information source. The touchscreen, camera and microphone input UICs were implemented following the observer pattern.

The last step is a modification of the application user interface presented to a user into a format that is device-independent. The application uses only graphical UI (GUI), which is simple enough to employ one of model-driven UI approaches, but for this experiment, it was implemented using the second approach. The final GUI is generated by the application itself according to capabilities of an output interaction channel (i.e., a device screen). The capabilities of the interaction channel necessary for GUI generation are device screen width and height in pixels and screen density in pixels per inch. The information provided as metadata of the output UIC service allows the application to render an appropriate GUI view and send it to the visual output UIC in a form of a GIF image (GIF provides best compression for this type of graphics). Using the same set of graphical resources and fonts as in the native application, it is possible to provide a pre-rendered GUI that mirrors the GUI of the original application.

The usage of the application involves a number of interactions. Assuming that the application is initiated from a default launcher application, the first interactions are the following:

1. The application gathers information on services exposed by the end-device used to initiate it and decides whether all required services are available. Almost all modern smartphones and tablets are equipped with a screen, geolocation, touchscreen, camera and microphone, so each such device is compatible.
2. The application retrieves capabilities of the visual output UIC, renders the main screen and sends it to the visual output UIC service. At the same time, the application registers as an observer using the touchscreen service.
3. The visual output UIC service running on the device displays the received image on the screen.
4. When the user touches the screen, the event is captured by the touchscreen input UIC service and the application is notified.
5. The application decides if the touch event corresponds to any actions, renders a new or updated GUI and sends it to the visual output UIC. For example, if the user touched the ‘See quests…’ button on the main screen, the application renders the Quests screen and sends it back to the device.

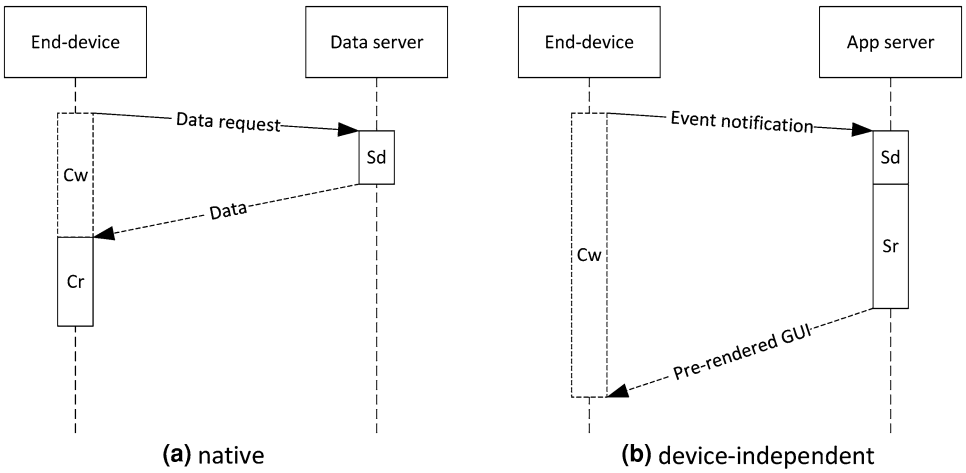

Fig. 4 Sequence diagrams of GUI state change in both versions of the application

4.2 Performance comparison

The internal logic of the application, the data source and client–server communication overheads are comparable in both versions of the application. Therefore, the main issue that could deteriorate the usability of the device-independent version of the application is GUI response time. It could worsen due to server-side GUI generation and increased amount of data that have to be transferred to handle a GUI state change. Taking into account a typical interaction loop (cf. points 3–5 of the interaction sequence presented in 4.1), the GUI response time measurement begins with a user action that causes a GUI state change and ends with a presentation of a new GUI view on the screen. Sequence diagrams with communication delay for both versions of the application are presented in Fig. 4.

In both versions of the application, the total GUI response time is composed of time required to prepare data (Sd), time required to generate and render the GUI (Cr, Sr) and communication delay. The 'Cw' is an idle state in which the application waits for a server response. The communication delay of the initial request and data preparation time are comparable in both cases and can be omitted. Therefore, to show how the device-independent approach increases the GUI response time, it is enough to compare GUI generation times and response transfer times.

To test this issue in a controlled environment, both versions of the application were executed on a MS Windows Phone 7 emulator provided by MS Visual Studio 2010 Express for Windows Phone. The delay introduced by the communication channel was estimated by calculating time required to transfer exchanged data through typical mobile communication channels such as GPRS (assumed throughput 56 Kbps), EDGE (236 Kbps) and HSDPA (7.2 Mbps).

Results of the GUI generation measurements are presented in Fig. 5. Average GUI generation time for the native application version is 9.3 ms and for the device-independent version is 45.7 ms. The increase is 36.4 ms.

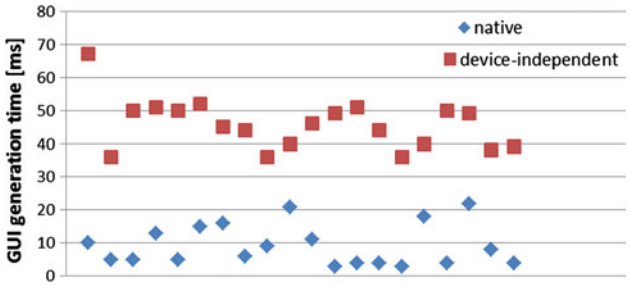

Fig. 5 Sequence diagrams of GUI state change in both versions of the application

To estimate the delay introduced by the communication channel, it was necessary to measure the size of data transferred from the server to the application. The size of data depends on which application view is presented. Table 1 presents average data sizes for each view of the prototype application and estimated delays for different channel speeds.

4.3 Summary

The presented experiment shows that applications implemented according to the DIA may indeed maintain full functionality and usability of a native application. All functions of the original application were recreated in the device-independent version of the application, and the UI of the application remained unchanged. The results of performance evaluation show that the device-independent version of the application requires more time to provide a new GUI view after a state change. The delay caused by server-side GUI generation is very small, but the device-independent version is more susceptible to communication channel throughput. Nevertheless, with broadband mobile communication channels, the total delay is well below 1 s,

Table 1 Results of the communication delay estimation

	Native				Device independent			
	AVG (Kb)	GPRS (ms)	EDGE (ms)	HSDPA (ms)	AVG (Kb)	GPRS (ms)	EDGE (ms)	HSDPA (ms)
Credibility stack	0.40	7	2	<1	34.83	622	148	5
Quests list	0.94	17	4	<1	43.54	778	184	6
Quest details	33.46	598	142	5	75.24	1344	319	10

which is assumed to be a reasonable limit for a static GUI response time [31]. Therefore, the overall application usability is not deteriorated by the increased GUI response time.

5 Conclusions and future research

The DIA, presented in this paper, is a response to the opportunity provided by the multitude of smart end-devices that compose the IoT—an opportunity that can result in development of truly device-independent and ubiquitous applications. The main idea behind the DIA is to move the processing out of end-devices to a cloud-based infrastructure and to introduce a set of protocols and services that separate applications from end-devices and provide support for development of multi-device applications. The presented experiment confirms that the DIA is a viable concept and that it provides the assumed benefits. The original application analyzed in the experiment was limited to smartphones running the MS Windows Phone 7 OS and the transformation into a device-independent application according to the DIA concept made the application available on any end-device featuring the set of necessary, but common, features: a screen, a geolocation service, a touchscreen, a camera and a microphone.

The DIA approaches the problem of application device independence from a new perspective and opens a wide range of new research topics. From provision of continuous and stable device connectivity, to detection of device availability crucial for multi-device usage scenarios, to UI abstraction, adaptation and distribution issues.

The presented DIA is an extensible solution and enables enhancements that provide additional functions. This creates an opportunity for new services that would help better support device-independent and multi-device applications: application markets and catalogs, private application repositories, third party services for gathering and sharing user preferences, device usage billing and micropayment services (provided, for example, by telecom operators), etc.

Each of these topics is a challenging task on its own. To succeed in this broad field, a coordinated research effort is required. We believe that the concept presented in this paper provides a solid framework for future research in the field of device-independent applications and will eventually ease the burden of developing truly ubiquitous applications that efficiently use capabilities of multiple devices.

References

1. Weiser M, Brown JS (1996) Designing calm technology. PowerGrid J 1(1):75–85
2. Weiser M (1991) The computer for the 21st century. Sci Am 265(3):94–104
3. European Commission (2009) Internet of things—an action plan for Europe. Communication from the Commission to the European Parliament, the Council, the European Economic and Social Committee and the Committee of the Regions, Brussels. http://ec.europa.eu/information_society/policy/rfid/documents/commiot2009.pdf. Accessed 15 Oct 2012
4. Zappa M (2012) Envisioning emerging technology for 2012 and beyond. Envisioning technology. http://envisioningtech.com/envisioning2012/. Accessed 15 Oct 2012
5. Cisco (2012) Cisco visual networking index: global mobile data traffic forecast update 2011–2016. Cisco White Paper. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.pdf. Accessed 15 Oct 2012
6. Voorhies D, Kirk D, Lathrop O (1988) Virtual graphics. ACM SIGGRAPH Comput Graph 22(4):247–253
7. Winokur D (2011) Flash to focus on PC browsing and mobile apps; adobe to more aggressively contribute to HTML5. Adobe website. http://blogs.adobe.com/conversations/2011/11/flash-focus.html. Accessed 15 Oct 2012
8. O'Reilly T (2007) What is web 2.0: design patterns and business models for the next generation of software. Commun Strateg 1:17–37
9. Prehofer C, van Gurp J, di Flora C (2007) Towards the web as a platform for ubiquitous applications in smart spaces. Second workshop on requirements and solutions for pervasive software infrastructures (RSPSI), Ubicomp 2007
10. Taivalsaari A, Mikkonen T, Ingalls D, Palacz K (2008) Web browser as an application platform: the lively Kernel experience. Technical report. Sun Microsystems, Inc., Mountain View, CA, USA
11. jQuery (2012) http://jquery.com/. Accessed 15 Oct 2012
12. Webinos project (2012) http://webinos.org/. Accessed 15 Oct 2012
13. MyMobileWeb project (2012) http://mymobileweb.morfeo-project.org/. Accessed 15 Oct 2012
14. Apple (2012) Thoughts on flash. Apple website. http://www.apple.com/hotnews/thoughts-on-flash/. Accessed 15 Oct 2012
15. Neves PA, Rodrigues JJ (2010) Internet protocol over wireless sensor networks, from myth to reality. J Commun 5:189–196

16. Erl T (2004) Service-oriented architecture. Prentice-Hall, Englewood Cliffs
17. Lawton G (2008) Developing software online with platform-as-a-service technology. Proceedings of IEEE Computer 2008, pp 13–15
18. Birman K, Joseph T (1987) Exploiting virtual synchrony in distributed systems. Proceedings of the eleventh ACM symposium on operating systems principles, SOSP'87, pp 123–138
19. Simon L, Mallya A, Bansal A, Gupta G, Hite TD (2005) A universal service description language. Proceedings of the IEEE International Conference on Web Services, ICWS '05. IEEE Computer Society, Washington, DC, USA, pp 823–824
20. Gamma E et al (1995) Design patterns, elements of reusable object-oriented software. Addison-Wesley, Boston
21. Chmielewski J, Walczak K (2013) Application architectures for smart multi-device applications. Proceedings of the Workshop on Multi-device App Middleware 2012, (Montreal, Canada, December 3–7, 2012), ACM, New York, pp 5:1–5:5
22. Calvary G, Coutaz J, Bouillon L, Florins M, Limbourg Q, Marucci L, Paternò F, Santoro C, Souchon N, Thevenin D, Vanderdonckt J (2002) The CAMELEON reference framework. Cameleon project deliverable. http://giove.isti.cnr.it/projects/cameleon/pdf/CAMELEON%20D1.1RefFramework.pdf. Accessed 15 Oct 2012
23. Serenoa project (2012) http://www.serenoa-fp7.eu/. Accessed 15 Oct 2012
24. W3C Model-Based UI Working Group (2012) http://www.w3.org/2011/mbui/. Accessed 15 Oct 2012
25. Cantera JM, Vanderdonckt J, Paternò F, Palanque P (2010) EICS panel: model-driven engineering of UIs: fact or fiction. Engineering Interactive Computing Systems (EICS 2010, Berlin, Germany). http://mymobileweb.morfeo-project.org/archives/model-driven-engineering-of-uis-fact-or-fiction. Accessed 15 Oct 2012
26. Perlman S (2010) Introducing the OnLive game system. OnLive Blog. OnLive. http://blog.onlive.com/2010/11/17/introducing-the-onlive-game-system/. Accessed 15 Oct 2012
27. Bule G (2012) OnLive cloud gaming technology for hosted virtual desktops? Virtual Desktop Persiflage Blog. tuCloud. http://tucloud.com/BLOG_onlive_powers_hosted_virtual_desktops.html. Accessed 15 Oct 2012
28. Gaikai (2012) http://www.gaikai.com/qa. Accessed 15 Oct 2012
29. Doan A, Ramakrishnan R, Halevy AY (2011) Crowdsourcing systems on the world-wide web. Commun ACM 54(4):86–96
30. Deterding S, Dixon D, Khaled R, Nacke L (2011) From game design elements to gamefulness: defining "gamification". Proceedings of the 15th International Academic MindTrek Conference, pp 9–15
31. Nielsen J (1993) Response times: the 3 important limits. Jakob Nielsen's Alertbox. http://www.nngroup.com/articles/response-times-3-important-limits/. Accessed 15 Oct 2012