Journal of Web Engineering, Vol. 10, No. 3 (2011) 234–272

MIGRATING DESKTOP APPLICATIONS TO THE INTERNET: A NOVEL VIRTUALIZATION PARADIGM BASED ON WEB OPERATING SYSTEMS

FABRIZIO LAMBERTI
Dipartimento di Automatica e Informatica, Politecnico di Torino
C.so Duca degli Abruzzi 24, 10129, Torino, Italy
fabrizio.lamberti@polito.it

ANDREA SANNA
Dipartimento di Automatica e Informatica, Politecnico di Torino
C.so Duca degli Abruzzi 24, 10129, Torino, Italy
andrea.sanna@polito.it

Received June 17, 2011
Revised September 5, 2011

In the past years, many solutions for virtualizing desktops and applications have been proposed. Unfortunately, given their significant resource requirements, their limited portability, and the achieved performances in terms of interactivity and usability, they did not prove to be capable of effectively replacing traditional local desktops. Recently, Web Operating Systems (Web OSs) started to be developed as an alternative approach for the creation of personal desktop environments, where newly designed applications created by leveraging on Web technologies can be accessed by end-users in a unified and seamless way. In this paper, a software architecture designed to further enhance the attractiveness of such environments by allowing existing desktop applications to be migrated into Web OS frameworks without any modification is presented. An automatic tool exploits image processing techniques to analyze the Graphics User Interface (GUI) of a remotely running application and to produce a detailed description for it, by recording its visual appearance and dynamic behavior. Then, this description is reloaded by a Web OS module that exploits remote computing techniques to provide the user with a local-like interaction with the virtualized application running on a remote machine. Thanks to the achieved separation between application logic and interface, the designed approach makes it possible to recreate virtual copies of original applications tailored to user device's characteristics, and it is additionally capable of providing significant improvements in terms of bandwidth usage and interactivity degree. Thus, without any re-coding, the original Web OS environment can be effectively enriched by letting the users run possibly customized copies of the same applications their are used to work with on a traditional desktop.

Keywords: Web OS virtualization, distributed and remote computing, web applications, user interfaces, software reusability
Communicated by: D. Schwabe & H. Rossi

1 Introduction

In the early stages of the computer age, the global scenario was characterized by a functional model in which monolithic mainframes were accessed by a number of users physically

located within the same facility to execute multiple computing tasks in a sequential way. When networking began to be developed, this centralized approach started switching to a novel paradigm, in which specialized text terminals (or "thin" terminals), endowed with simple data entry and display capabilities, were used for allowing multiple users to share the processing power of a remote server machine [1, 2]. With the introduction of Graphics User Interfaces (GUIs) and of window-based operating systems, this scenario changed again, and text terminals progressively evolved into "rich" (or "thick") terminals capable of delivering to the user a more attractive visual experience. Graphics desktops started to be developed, allowing users to locally run a large set of complex applications by mostly exploiting client-side processing capabilities, thus achieving a much more responsive interaction [3].

However, in more recent years, the widespread diffusion of Internet technologies brought back the interest to the thin client-server model, which is today often associated with the cloud computing paradigm. In this evolved scenario, some desktop applications can be characterized by a local GUI that interacts with a server-side logic running "in the cloud" for carrying out its processing tasks [4, 5]. The extremization of this paradigm resulted in the definition of a virtualization model in which the whole desktop can be possibly replaced by a virtual environment controlled by remote: in this case, the client only acts as a presentation device, being in charge of handling user interaction and of displaying processing results as a flow of screen updates generated on a server machine [6, 7, 8]. In the last decade, many solutions and frameworks for virtualizing complete computer desktops and/or their applications were developed, like the X Window system [9], the Virtual Network Computing paradigm [10], and the Remote Desktop technology [11]. The reasons motivating the research efforts in this field were manifold: on the one hand, virtualization could allow to reduce the cost of hardware required on local clients, since heavy computing operations could be executed on high-end server machines. On the other hand, resource usage could be optimized, controlled, and balanced in a more efficient way. Moreover, software centralization could result in a simplification of maintenance and administration procedures. In fact, application logic could be upgraded in a transparent way without any modification to software installed on client machines. Furthermore, data could be secured in a more efficient way, by adopting centralized malware protection, anti-theft, redundancy, and backup techniques. Finally, virtualized desktops could be accessed by multiple locations, thus allowing users to work into a true nomadic scenario where they could have seamless and continuous access to their own applications and related data [12].

Despite the benefits that could be introduced by the adoption of such virtualization approach, existing implementations did not prove to be capable of behaving as ultimate remote desktop solutions. Thus, these virtualization techniques have been mainly exploited as a mean for occasionally controlling remote environments and applications. In fact, modern users are accustomed to having at their disposal high-responsive desktop environments, and they would not be easily interested in exchanging them for a possibly slow and frustrating remote experience over congested network connections. Moreover, most of the above virtualization architectures have specific requirements on client and server software configurations. This limits the practicability of a truly ubiquitous and cross-platform access, and further reduces benefits possibly associated with the replacement of a local desktop with a remote one [13].

Recently, with the further evolution of cloud computing, Web Operating Systems (Web

OSs) started to be developed as an alternative to current virtualization solutions [14, 15]. Like the above remote computing techniques, Web OSs are aimed at providing a feasible local desktop replacement strategy. However, being based on the Web, they are able to offer a wider portability across multiple heterogeneous platforms, today ranging from ad hoc mobile computers (like the Google Chromebooks, http://www.google.com/chromebook) to home appliances and other consumer devices [16]. By relying on modern Web technologies, they allow for the deployment of attractive Web applications, in which graphics interfaces can be managed on the local platform, while application tasks can be either executed using client-side or server-side processing techniques [17, 18]. Through the adoption of several network and local caching techniques, some Web OS implementations also allow to further reduce the amount of bandwidth required, enabling in some cases offline operations. According to Web OS developers, the success of this novel paradigm will strictly depend on the richness of the application set that will be made available to the end-users [13, 19]. Ideally, this would require the migration to Web OS platforms of all those applications the users are accustomed to run on their traditional desktops. Unfortunately, even though today's Web OS solutions come with powerful GUI toolkits enabling in some cases the construction of complex applications (including word processors, spreadsheets, media players, etc.), new applications still need to be developed from scratch, with the consequence of critical time and money losses.

In this paper, a virtualization paradigm aimed at addressing the above requirements by allowing existing desktop applications to be integrated into available Web OS environments without any re-design is proposed. The designed architecture builds upon the work presented in [20], where a preliminary framework for transferring application GUIs to mobile devices is defined. Being based on a blend of remote computing-oriented approaches and of Web-related technologies, the proposed solution represents a feasible strategy for transparently extending the native set of Web OS applications, by simultaneously preserving the characteristic portability of the Internet scenario, and fostering the reusability of existing software. Additionally, extensive experimental tests showed the possible advantages of the proposed architecture with respect to comparable approaches in terms of network resources usage.

This paper is organized as follows. In Section 2, a review of some of the main solutions enabling remote control of desktop applications is provided. In Section 3, the proposed Web OS-based virtualization approach is presented; moreover, an overview of the overall architecture is given, and the major advantages provided by the designed methodology are analyzed through a preliminary comparison with alternative techniques. In Section 4, the main blocks of the designed architecture are analyzed in details from a functional point of view. The designed image-based classification rules and the strategies developed for handling interfaces' dynamic "behaviors" are presented in Section 5. An overview of client-side and server-side software developments is reported in Section 6, together with qualitative observations. Finally, results of experimental tests aimed at evaluating virtualized GUI accuracy, as well as at profiling the network overhead and the degree of interactivity are analyzed in depth in Sections 7 and 8, respectively.

2 Remote Computing and Virtualization Techniques

One of the first architectures designed to take into account the possibility of remotely controlling graphics applications is represented by the X Window system, or X [9]. Basically,

software platforms based on X – like Unix and Linux distributions – allow a graphics interface (the X client program) to be displayed on a remote machine (running the X server) in a seamless way. X clients and servers exchange X protocol messages consisting of low-level primitives for creating windows, managing user inputs and drawing lines, curves and bitmaps, with the aim of giving end-users the impression of a co-located interaction.

The remote control and visualization philosophy of X has been adapted and extended in many ways. One of the clearest adaptation examples is given by Virtual Network Computing (VNC). VNC [10] refers to a desktop sharing technology that allows users to remotely control a computer using a specific protocol named remote framebuffer (RFB). Through RFB, keyboard and mouse events can be transmitted from one computer to another, together with small rectangles extracted from the remote machine's framebuffer. The client, often referred to as VNC viewer, is simply in charge of handling user interaction and of redrawing the display of the remote computer; on the other hand, the server is responsible for extracting graphics contents from the framebuffer and for transmitting them to the client machine. VNC works at an higher level than X, thus achieving larger portability. In fact, VNC is available on almost all the current desktop operating systems, and there also exist VNC client implementations based on Java, Microsoft ActiveX, Macromedia Flash, etc. [21]. However, these implementations are often characterized by poorer performances, and they are not always capable of providing the whole set of features available in the corresponding desktop versions (for instance, they do not always allow the user to take control over a single "logical" application).

Another virtualization solution evolved from the above basic idea is represented by the Remote Desktop technology [11]. Remote Desktop relies on RDP (Remote Desktop Protocol), a protocol developed by Microsoft that allows a given user to log into a remote machine equipped with MS Terminal Services (TS) and to control it by remote [23, 24]. Being tailored to a specific operating system, a RDP server is able to gain a higher control on the local machine, allowing remote users to access not only graphics resources, but also audio drivers, printers and storage devices. Recently, Microsoft extended the original Remote Desktop philosophy by introducing the RemoteApp modality [25], that allows users to add a single remote application running on a Windows Server 2008 machine to their local desktop. Remote Desktop and RemoteApp functionalities can be made accessible through a Web browser by using the TS Web Access ActiveX client. Thanks to its native integration in the host operating system, Remote Desktop technology is able to achieve better results with respect to previous solutions.

Unfortunately, neither Remote Desktop nor the other techniques discussed above are capable of providing users with a complete and effective local desktop replacement solution [13]. This is mainly due to their implicit limitations in terms of portability and usability, to their possibly constrained performances (e.g., due to network overheads), and to the difficulties associated with the creation of a unified desktop environment.

A different approach to get access to a remote desktop (and to its applications) is represented by Web Operating Systems [14, 26, 27, 28, 29]. A Web OS is a sort of virtual environment that runs within a Web browser and provides access to a set of applications, thus mimicking a traditional desktop operating system. Web OS solutions rely on the so called Web 2.0 [30], i.e., a set of technologies like HTML5, AJAX (Asynchronous JavaScript and XML), Flash, and Java that constitute the basic building blocks for the construction of

Web applications [17]. Web OSs, as Web applications, simply rely on the availability of a network connection and of a thin client equipped with an Internet browser. Before the advent of Web OSs, many examples of Web applications had been developed, sometimes grouped into a unified Web "container" (e.g., GoogleDocs, http://docs.google.com). The main limitation of these applications is that they did not offer the whole set of functionalities available in traditional desktop environments. Moreover, they were not able, by themselves, to recreate the appearance and usability of a common (and integrated) window-based desktop. Thus, Web OSs began to be developed.

Many examples of both commercial and open source Web OSs are already available. They can be classified in three main categories: virtual desktop-based Web OSs, mixed Web OSs, and "true" Web OSs. Virtual desktop-based Web OSs rely more on VNC and Remote Desktop technologies rather than on true Web applications, and require specific client software to be installed on the local machine. An example is DesktopOnDemand (http://desktopondemand.com), that provides X Window-based access to a customized Linux desktop embedded with many common applications. A similar approach is represented by Nivio (http://www.nivio.com), a Web OS totally based on Remote Desktop. In Nivio, registered users can access a TS-based desktop, and can add necessary applications on a pay-per-use basis, by choosing them from a restricted pre-configured set. Another example is given by SUN Microsystems's Secure Global Desktop (http://www.sun.com/software/products/sgd), that provides a Web-based framework in which Unix and Windows applications can be accessed through separate windows of a Java-enabled browser. On the other hand, true Web OSs only make use of Web applications. Despite the increased portability guaranteed by their intrinsic Web nature, true Web OSs are often characterized by a constrained set of native applications. However, there exist some implementations, like for example G.ho.st (http://www.g.ho.st), that address this lack by integrating external Web applications. Lastly, mixed Web OSs represent an in-between solution with respect to the previous ones. They are built according to Web 2.0 philosophy, but they make up for the shortage of applications by partially relying on local software installed on the client machine. Thus, for instance, eyeOS (http://eyeOS.org), Chrome OS (http://www.google.com/chrome), Desktoptwo (http://desktoptwo.com), and YouOS (http://www.youos.com) can offer a PDF reader, a video player, and other applications supposed that suitable plugins are available into the Web browser. However, this latter strategy partially limits the portability of the resulting frameworks, by imposing additional requirements on the configuration of the local machine.

3 Web OS-based Virtualization

In this section, the basic idea behind the proposed Web OS-based virtualization methodology is presented; then, an overview of the designed architecture is reported; finally, a comparison between the devised approach and existing virtualization techniques is provided, and major innovations introduced by the present work are outlined.

3.1 Basic idea

In this paper, a technique for possibly supporting the transformation of today's Web OSs into suitable desktop replacement tools is presented; the devised approach consists in a unconventional virtualization paradigm that is capable of separating the program GUI from

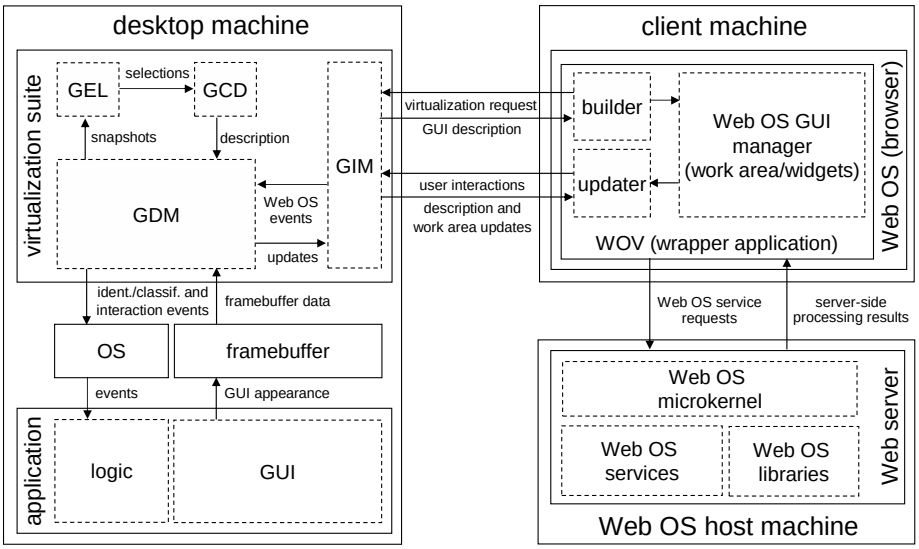

Fig. 1. Overall architecture of the proposed Web OS-based virtualization framework.

the application logic, thus allowing to effectively migrate existing applications running on traditional window-based desktops to Web OS environments, without any re-design or code re-writing. The designed architecture relies on a software framework firstly presented in [20], that is capable of analyzing the graphics interface of a desktop application, and of generating a structured description to be later reloaded for recreating the original GUI on a remote machine. The main aim of the solution in [20] was to transfer existing applications to mobile devices by maintaining their native look and feel (i.e., mimicking their original appearance, and fostering consistency, or "continuity"). In the current work, in turn, the goal is to migrate desktop application logic to the target environment, by "dressing" it with a new (and unified) graphics appearance. As it will be shown in the following, the above separation between application behavior and interface appearance allows to effectively integrate different technologies (i.e., Web, virtualization, cloud computing, and remote visualization) into an effective solution that is capable of tackling the main constraints of alternative approaches (like Remote Desktop, VNC, X Window and basic Web OS implementations). Lastly, it is worth observing that the main limitation of the approach in [20] was represented by the impossibility to manage GUI "dynamic" changes occurring during remote interaction (e.g., associated with appearing, disappearing, and changing graphics elements). The above events are very frequent in today's graphics interfaces; thus, this lack severely restricted the concrete application fields for such a preliminary framework. This limitation, as well as the above considerations were taken into account in designing the architecture discussed in this paper.

3.2 Overall architecture

The overall structure of the designed architecture is summarized in Figure 1. On the server side, a *GUI Dynamic Monitor* (GDM) is in charge of analyzing the interface of the application that has to be virtualized, and of producing a description of its constituting graphics

elements (or widgets) and work area(s). To accomplish this task, the GDM module relies on a *GUI Element Locator* (GEL) that analyzes the portion of the framebuffer corresponding to the application itself, and extracts images (referred to as *selections*) corresponding to possible isolated interface elements. These selections are then passed to a *GUI Classifier and Descriptor* module (GCD) that actually assigns each selection to a particular widget type and records this information into an interface description file. On the client side, the user can run a *Web OS Virtualizer* (WOV) software module to request the execution of a particular remote application. The WOV downloads the application description file and automatically reconstructs the interface of the desktop application by exploiting the graphics primitives made available by the local Web OS GUI toolkit. Thus, the WOV acts as a sort of application wrapper capable of recreating any application exported by the remote server.

User interactions with the local interface (i.e., mouse clicks, key presses, etc.) are captured by the wrapper and transferred to the *GUI Interaction Manager* (GIM) running on the server side, that is responsible for their conversion into suitable events to be passed on to the operating system queue. The GDM continuously monitors possible variations in user interface's appearance that could result from the processing of the received events, and notifies the WOV application of these changes. In particular, events resulting in a modification of the application GUI (e.g., the appearance of new elements, the opening of a popup window, etc.) force the GDM to activate the GEL and GCD modules, which automatically update the interface description file and pass it to the Web OS wrapper. On the other hand, events that directly affect the work area(s) result in the extraction of the corresponding graphics regions from the framebuffer and in their transmission to the Web OS application as a flow of graphics updates (through remote visualization techniques). Widget and work area updates allow the WOV to maintain the visual synchronization between the status of the local and the remote GUIs, thus giving to Web OS users the impression of a local-like interaction with the remote application.

3.3 Advantages of the proposed solution compared with existing approaches

The major innovation of the proposed approach consists in the definition of a mechanism for extracting a structured description of the graphics elements constituting the interface of a common desktop application. In this way, a neat distinction between application logic and graphics interface can be identified, even when this is not made evident in the native environment. As shown in this paper, the above distinction can be effectively exploited for designing a novel virtualization paradigm based on Web OS technologies, even though other application scenarios could be also envisaged. In the following, the main advantages of the proposed virtualization technique are discussed, and a comparison between Web OS-based virtualization and already mentioned alternative solutions is provided. Findings resulting from the above comparison are summarized in Table 1.

- *Server-side platform neutrality*: the proposed approach relies on a novel method for parsing the GUI of a remotely running application and for describing its constituting elements by means of cross-platform image processing techniques. By exploiting the above technique in the context of virtualization, the designed strategy allows existing applications running under different operating systems to be executed (possibly at the same time) within the same Web OS environment. This goal cannot be achieved with

Table 1. A comparison of the designed Web OS-based virtualization approach with alternative remote computing techniques and basic Web OS solutions.

Remote Desktop, VNC, X Window	**Basic Web OSs**	**Proposed Web OS-based virtualization approach**
Rich set of applications	New applications developed from scratch by exploiting different Web technologies	Native application set enriched by embedding virtualized copies of desktop applications without any re-design or code re-writing
Full desktop or single application virtualization	Oriented to full desktop replacement	Complete desktop environment: native Web OS applications side-by-side to virtualized applications
Delivery of image-based representations of the whole interface (Remote Desktop, VNC) or transmission of graphics primitives, including drawing commands and raw image data, used for refreshing the virtual frame buffer (X Window): no separation between updates to GUI and work area	Client and server side computations with lower bandwidth requirements	Separation between application work area and GUI widgets: GUI locally managed with reduced network overhead
Need for a complete desktop environment/for the installation of specific software (on the client side) and for a particular platform, e.g., a Unix-like operating system (on the server side): limited portability	Supporting multiple server configurations, without requiring the installation of local software, except for browser plugins: oriented to software portability	Running in different server environments, without requiring the installation of local software: full application portability
Virtual applications running within a window of the local system/within a Web browser: poor integration with the appearance of the local desktop	Web desktop mimicking that of traditional graphics operating systems	Virtual GUI recreated using local Web OS toolkit used by the other (native) applications: homogeneity
Interaction techniques of native operating system	Web interaction	Web interaction plus other Human Computer Interaction techniques without any modification
Unmodified GUI of the native application (original application treated as an atomic entity)	Graphics interface designed ad hoc using Web OS GUI toolkit	Automatically generated GUI of the desktop application: GUI customization based on user needs and platform characteristics

X Window-based virtualization, as this approach necessarily relies on the presence of a Unix-like remote machine. A higher degree of freedom is guaranteed by Remote Desktop and VNC-oriented approaches, which can be exploited over a heterogeneous set of remote platforms; however, the general-purposeness of the above techniques is achieved at the cost of poorer performances as well as of a looser control on virtualization features.

- *Client-side portability*: being based uniquely on Web technologies, the proposed approach allows to create rich desktop applications by preserving their portability across multiple client devices as they may be executed without requiring any software installation. By contrast, when client side and server side do not share the same operating system, traditional virtualization solutions require, in general, the installation of sophisticated software on the client machine. This is even more true when Web-based virtualization scenarios, which are the actual focus of this work, are considered: in this case, most of the existing solutions require, at least, the existence of a local Java installation (since implementations that can be accessed through pure HTML5 and AJAX technologies are characterized by extremely limited performances). In turn, the proposed solution can be effectively run on a wide spectrum of Web-enabled platforms, not necessarily supporting the Java technology. As a matter of example, during the development phases, the designed solution has been successfully evaluated on a series of devices including Apple iPod/iPhone, Apple iPad, and Nintendo Wii.

- *Interface customizability/adaptability*: the devised virtualization solution relies on the availability of a description of the graphics elements constituting the interface of a remotely running desktop application. The above description can be possibly used to locally recreate a virtual instance of the application GUI by radically adjusting its graphics layout: thus, for instance, widget type may be altered, graphics elements may be moved from one container to another, new aggregations may be defined, etc. That is, users are allowed to create their own interfaces by using original GUI elements as functional (customized) building blocks. The possibility of adjusting the aspect of graphics elements can also be exploited to tailor GUI appearance and behavior to the particular input modalities available on the selected virtual platform. As a matter of example, sliders, menus and combo boxes that may be difficult to control using fingers (e.g., on touch-enabled mobile devices) or using alternative pointing devices with reduced accuracy (like, for instance, the Wii Remote controller) could be replaced by more "comfortable" buttons of a suitable size organized in a proper way. In summary, with respect to existing virtualization solutions like Remote Desktop, VNC and X Window, the devised approach is able to effectively adapt to both user preferences and device characteristics.

- *Application accessibility*: thanks to the knowledge of the exact location and behavior of the elements used in the graphics interface of a given software program, the proposed virtualization approach has the additional advantage of making it possible to seamlessly introduce in the application, besides native input/output modalities, new Human Computer Interaction (HCI) techniques, even when this possibility was not foreseen in the original desktop environment. Thus, while in the existing virtualization approaches the original interaction modalities are simply "exported" to the client side, the designed solution natively allows to augment software accessibility without requiring any modification to the remote desktop application.

- *System integration transparency*: in the existing Web-enabled virtualization approaches, virtualized applications are either characterized by the original graphics appearance (e.g., with Remote Desktop or VNC) or by the look and feel provided by the virtual

graphics toolkit, not necessarily aligned with that of the local operating system (e.g., AWT or Swing for Java-based X Window applications). On the contrary, in the proposed approach, virtualized applications can be recreated with the same graphics appearance of the host Web OS framework and used side-by-side with native applications of the selected Web environment. In this way, a seamless integration within a homogenous portable desktop environment is transparently achieved. It is worth saying that the completeness and fidelity of such an integration (as well as many of the other advantages of the proposed approach) are strongly related to the "quality" of the interface description generated at the server side. For this, an in depth analysis of aspects related to the accuracy of localization and classification algorithms is provided in Section 7.

- *Network efficiency*: the creation of a clear separation between application logic behavior and user interface appearance additionally lets the proposed approach achieve improved performance with respect to alternative virtualization approaches, especially when intensive GUI applications are considered. In fact, in the Remote Desktop and VNC-oriented approaches, graphics blocks are copied from the desktop frame buffer to the virtualized frame buffer, and changes occurring to the application GUI are handled together with changes affecting the work area(s); thus, as a matter of example, when a menu is closed, the application area previously obscured has to be retransmitted even if no changes have occurred. On the contrary, even though the designed solution uses a similar approach to deal with work area updates, GUI interactions are locally managed at the client side; this allow to achieve improved performances with respect to the above techniques in terms of both bandwidth usage and interaction latency thanks to the reduced data flow. Things are slightly different when the X Window-based virtualization technique is considered; in this case, the X client running on the desktop machine sends graphics primitives (including image raw data) to the X server at the client side, and directly performs low-level drawing operations in the virtual frame buffer. The above steps are performed in a more efficient way with respect to Remote Desktop and VNC-based solutions, and a relevant saving on network resources can be achieved. Nonetheless, client-server synchronization during GUI interactions still results in the exchange of a significant amount of graphics data (in both the downlink and uplink directions). This is due to the fact that X Window only partially exploits its knowledge of GUI and work area peculiarities; thus, manipulations on interface elements (e.g., inspection of menus, opening of new windows, etc.) require the transmission of drawing information needed to refresh the display that are equivalent to those used to update the work area(s). On the other hand, even though the proposed approach makes a comparable use of graphics data exchanges during operations affecting work area(s), the amount of information transmitted during GUI interactions is significantly reduced, thanks to the link that is established between GUI elements and application behavior. Based on the above considerations, by using the proposed approach a performance improvement associated to a thick client-based model can be expected. Results of experimental tests aimed at carefully profiling system behavior in terms of bandwidth occupation and interaction latency are discussed in Section 8.

4 Core Functional Blocks of the Proposed Architecture

In this section, the Web OS-based virtualization architecture discussed in this paper is analyzed from a functional point of view, and the main research issues addressed are pointed out. The discussion moves from the assumption that, as illustrated in Section 3, effective Web-based desktop replacement environments could be created once a structured description of applications commonly used in today's operating systems is made available, as this would allow such applications to be "moved" on the Internet in a seamless way. Thus, aspects related to image-based GUI elements localization and identification are first considered. Then, an analysis of the designed element classification and GUI description steps is provided. Finally, issues associated with the reconstruction of the original interface within a generic Web OS environment are discussed.

4.1 Identifying GUI elements

Previous works [20, 31, 32, 33] showed that extracting a description of the elements constituting the interface of a graphics application (whose source code is not available) is a task that cannot be completely accomplished by simply relying on image processing techniques. In fact, interface appearance usually depends on the particular GUI toolkit being adopted for designing the application layout. Moreover, even though discovering, for instance, the presence of a combo box can be achieved on almost all the platforms by trivial pattern recognition methods (e.g., configured for localizing the traditional down arrow icon), the identification of more complex elements requires to deal with the inner behaviors of graphics elements themselves (activated, for instance, by mouse clicks, button presses, etc.). In summary, an accurate interface description strategy cannot be pursued without interacting with the logic that lies behind the application itself. Several graphics-unaware alternatives, relying on reverse engineering techniques [34, 35], or on accessibility/automation features explicitly exposed by application interfaces through native graphics toolkit libraries [36], exist. However, on the one hand, the former methods could often lead to incomplete results; on the other hand, the latter approaches could theoretically achieve a complete understanding of application GUI, but their performances are strongly related to the completeness of the underlying libraries and to designers and programmers' choices.

Nonetheless, it is worth considering that modern GUIs actually embed "hidden" information that can be effectively exploited to support the element localization and identification step required by the proposed approach. In fact, besides reacting to user interaction with common behaviors (like moving the selection to another radio button, or opening the drop-down list of a combo box), user interfaces of today's window-based operating systems are designed to drive user attention through many visual signs. For example, when the mouse is moved over an element of the interface, this element is highlighted, i.e., it is bounded with a colored box, it is drawn with a brighter color, or it achieves a 3D aspect. Moreover, when the mouse is moved over a text portion that can be edited, the mouse pointer's aspect is varied to signal this possibility to the user. Independent of the specific effect, these behaviors implicitly notify the presence, as well as the precise position, of the interface element itself.

In the considered architecture, image processing and pattern matching techniques have been integrated with implicit visual information that can be displayed by the interface to let the GUI itself participate in the element identification (and classification) step. This allows

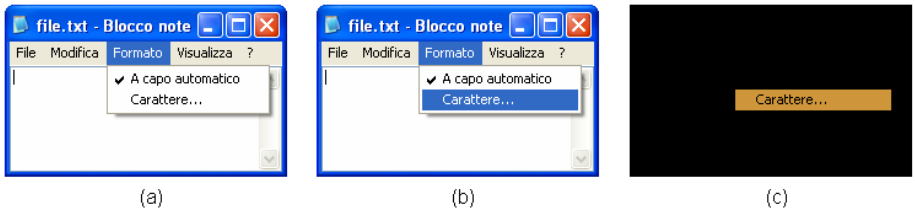

Fig. 2. User interface signalling the presence of a graphics element: (a) original GUI representation, (b) highlighting, and (c) difference image.

to improve the accuracy of the interface description results, and to achieve a higher degree of portability across various platforms with heterogeneous GUI "styles". In the GUI element identification procedure, the GDM module mimics user interactions by automatically moving the mouse (through system calls) over imaginary horizontal lines spanning the whole interface, thus forcing the interface to generate highlighting and mouse pointer updates. The GDM extracts from the framebuffer the graphics representations of the interface before and after the event (both with and without the mouse pointer), and passes them to the GEL module. In the difference image obtained by computing the exclusive-OR between the two images not including the mouse, pixels corresponding to the highlighted element are non-black. Thus, a simple algorithm able to identify the boundaries of the non-black region provides the GEL with the precise location and extension of the highlighted element. Similarly, the image obtained by computing the difference between the framebuffer content (with the mouse) captured after the event and the interface representation (without the mouse) before the event gives the exact shape of the mouse. The combination of the above techniques allows the system to identify elementary widgets like buttons, check boxes, text fields, radio buttons, scroll bars, etc. The highlighting effect generated by a menu item and the corresponding difference image are illustrated in Figure 2. Some interface elements need an additional processing step requiring further interactions: for instance, the complete description of a menu or a combo box requires to generate a mouse click event, to compute the difference image, and to move the mouse over the various items included in the drop-down list (and this procedure has to be repeated for recursively nested subitems). As shown in Figure 3, the GDM is also able to automatically detect the application work area/s that has/have to be treated through remote visualization techniques. The GDM and GEL modules have been endowed with several wizards that possibly let the user participate in the various phases of the identification process. This allows, for instance, to adjust the size and location of the work area, thus providing a more precise control over the final GUI appearance, as well as over bandwidth requirements and interaction performances. As a matter of example, the results of the above processing steps applied to the identification of the Microsoft PowerPoint GUI elements are illustrated in Figure 4.a.

4.2 Classifying and describing GUI elements

The GCD module receives in input a list of selections identified by the GEL component. The GCD extracts the group of pixels included within the selection boundaries from both the graphics representation of the interface (where the element in the selection is not highlighted),

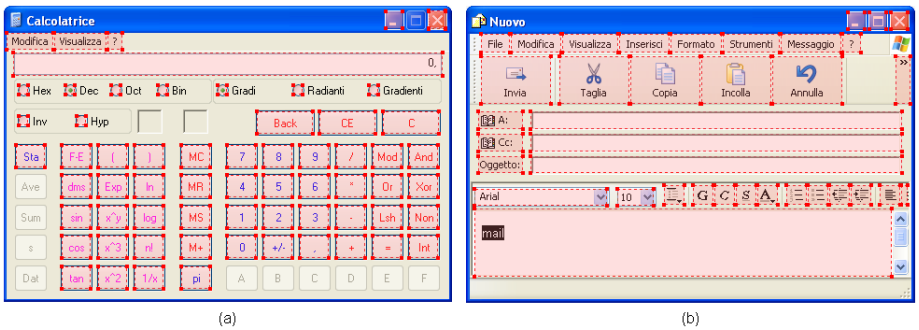

Fig. 3. Work area identification on (a) Microsoft Calculator and (b) Outlook Express interfaces.

and from the difference image. The first image is processed by ad hoc algorithms that compute essential information to be used in the classification procedure. Information above basically relies on the concept of block, that is a group of neighboring pixels sharing some common characteristics. Thus, for instance, a block could be one of the characters of a label, a button icon, or a combo box arrow. In a preliminary step, the number of generic blocks, the distance between blocks, the size of each block and its position with respect to the beginning and end of the selection, the number of icons, the number of arrows, and the specific arrow type are identified. On the other hand, the second image is analyzed to determine the particular type of highlighting occurred. Once all these details are available, several pattern-matching classification rules are applied. For sake of completeness, the classification rules designed for several common graphics elements like menus, combo boxes, radio button groups, text boxes, and text areas are illustrated in Section 5.

It is worth remarking that, while the widget identification phase has been designed to be as general as possible in order to adapt to various GUI styles, the present classification step is aimed at demonstrating the feasibility of the overall architecture, and does not claim to be comprehensive at all. Thus, more sophisticated (and possibly more general) solutions able to achieve a higher accuracy can be probably found for this step. However, results that can be achieved with the present implementation already allow the system to recreate extremely complex interfaces, and to effectively reuse into Web OS environments many everyday desktop applications. For example, in Figure 4.b, the results of the classification step over the selections of the Microsoft PowerPoint interface (Figure 4.a) are shown. Results achieved on a wider set of common desktop applications are additionally reported in Sections 7 and 8.

Once the graphics elements have been identified and classified, a description of the interface is generated and stored. Several User Interface Description Languages (UIDLs) [37] are available: examples include Plastic User Interfaces [38], UIML [39], XIML [40], UsiXML [41], TERESA XML [42], and many others. In the context of this work, the XUL format [43] (XML User Interface Language) has been chosen. XUL is a UIDL based on XML that allows to describe the visual appearance of a graphics interface through a predefined set of structured basic elements. Basic elements can be further extended by the user, and recursively nested to create more complex controls. Native XUL elements only allow to describe windows, buttons, menus, combo boxes, scroll bars, check boxes, radio buttons, text fields, labels, and images.

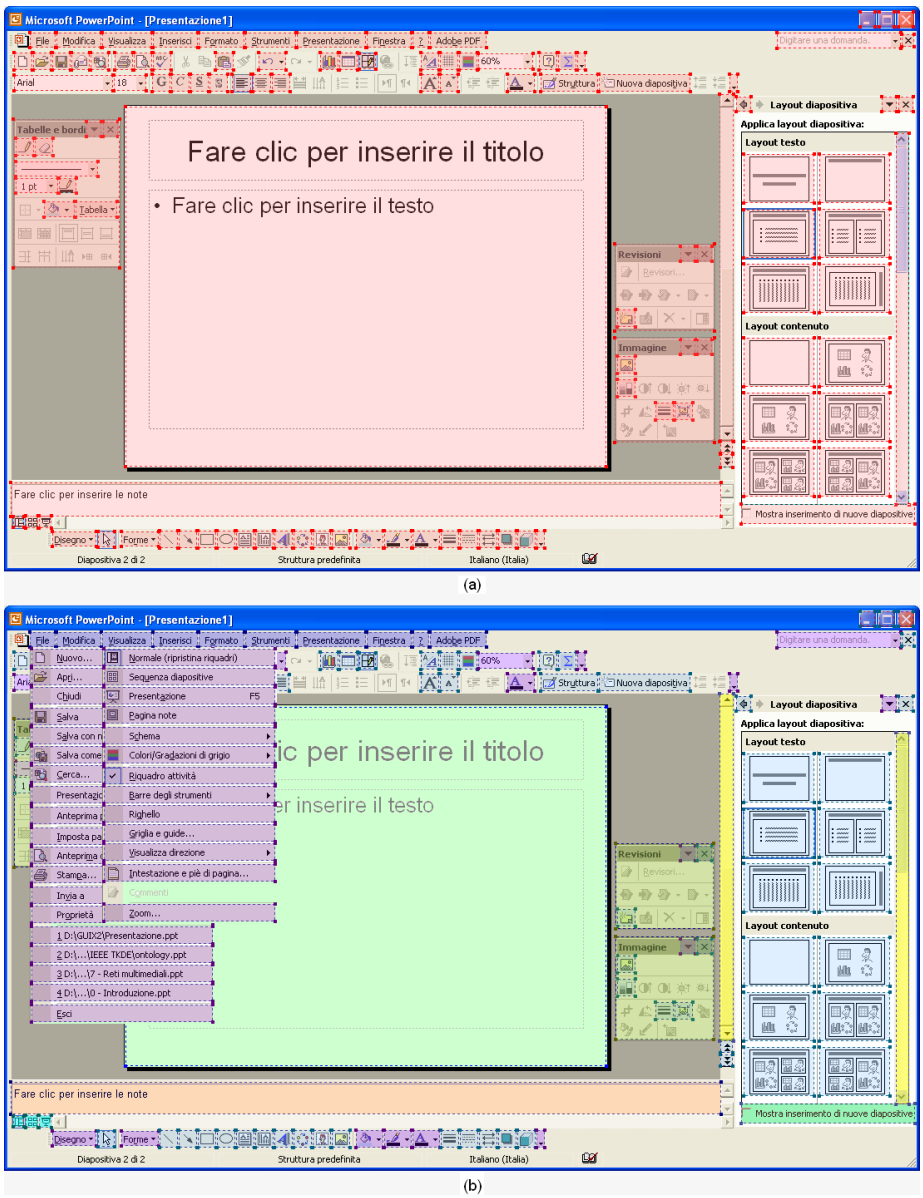

Fig. 4. Results of the GUI elements (a) identification and (b) classification steps over the Microsoft PowerPoint interface (under Windows XP).

Thus, basic XUL description capabilities have been enriched by adding, for instance, new elements for describing push buttons, list boxes, and work area(s). Moreover, native XUL elements have been extended with new attributes capable of capturing advanced graphics behaviors of modern interfaces. For instance, the combo box description has been allowed to manage tree-structured drop-down lists, while the window element has been redefined to distinguish among several possible window types. XUL is currently used to define the user

interfaces of most, if not all, Mozilla software products.

4.3 Recreating the remote GUI within a Web OS framework

On the client side, the user can run the Web OS wrapper for connecting to the remote server and for retrieving the list of installed applications. When the user picks up one application from the list, the virtualization framework automatically executes it on the server machine. If a description of the remote GUI is already available for the specific user/device, the GIM delivers it to the WOV application; otherwise, the GEL and GCD modules are activated, the graphics elements of the interface are classified, and a novel XUL description is generated on-the-fly. Information recorded in the description file is exploited by the WOV to locally recreate the appearance and behavior of the virtualized application. The WOV is constituted by three functional blocks, namely the *builder*, the *updater*, and the *manager*. The builder component is in charge of retrieving the GUI description transmitted by the GIM and of automatically recreating its constituting elements using locally available graphics widgets. Once the original structure of the remote interface has been recreated (possibly exploiting predefined mapping strategies specifically tailored to the specific virtualization device), the user can further tailor its appearance to his/her own needs by moving graphics elements from one container to another through intuitive context menus and drag and drop operations controlled by the manager component. Strategies for automatically adjusting interface appearance to match user and host environment constraints could be possibly integrated in this step [44, 45]. Structural modifications carried out on the interface are recorded into the XUL file associated to the selected user/device on the desktop machine. Thus, at his/her next access, the user will be provided with his/her updated custom interface. User interactions with the interface (i.e., with widgets and work area(s)) are monitored by the manager module, that is responsible for collecting mouse and keyboard events and for transmitting them to the remote GIM module through the updater. By exploiting the knowledge of the mapping between local and remote interfaces in terms of (possibly customized) element location and behavior, the server-side virtualization framework generates suitable events simulating the interaction with the local application, and inserts them into the host operating system's event queue. When processed by the system, these events can affect both application interface and work area(s). Events resulting into a dynamic modification of interface appearance (e.g., opening of a new window, appearance/enabling of previously hidden/disabled elements, etc.) activate the GDM, which automatically recognizes modifications in GUI aspect and/or functionality and updates the XUL description, accordingly. Similarly, the effects of events modifying the content of monitored work area(s) are identified, and they result in the production of the corresponding graphics representations with the updated framebuffer contents. Interface description and work area updates are continuously transmitted to the updater module, that transparently applies the necessary modifications to the Web OS application interface by interacting with the manager component.

5 Image-based Classification Rules

In the previous section, the overall strategy pursued by the proposed virtualization approach has been presented. In particular, the discussion focused on the mechanisms used for generating the initial interface's description (to be used for recreating the virtualized Web OS

application at system startup), and for managing updates interesting work area contents. In this section, several details on the image processing-based rules used for classifying GUI elements will be provided, by taking as examples some common widgets. Moreover, since in today's desktop applications user interactions could result in modifications directly affecting interface's appearance (i.e., the interface's behavior itself can be regarded as dynamic), the additional strategies that have been adopted to maintain a visual synchronization between local and remote GUIs in these situations will be analyzed.

5.1 Classifications rules for managing common GUI elements

In the following, the classification rules designed for dealing with some representative GUI elements like menus, combo boxes, text boxes, and text areas are illustrated. It is worth remarking that the presentation is not meant to be exhaustive, as additional classification rules would need to be defined in order to deal with other widget types possibly used in specific applications.

- *Menus*: menus are identified by exploiting their characteristic highlighting effect, that completely outlines their bounding boxes, inverting colors of both container and text. However, menu classification is a manifold process that also requires to navigate the complete tree hierarchy that is unveiled only when the menu is opened. Thus, the GDM is programmed to force menu expansion, and to move the mouse over all the menu subitems, thus generating the highlighting. For each highlighted region, the GCD extracts the icon and recognizes the text associated with the selected item (through OCR); it also determines whether the element corresponds to a leaf in the menu tree, or if it can be further expanded. In this case, a recursive procedure allows to explore the whole menu's graphics structure and to extract a complete description.

- *Combo boxes*: in modern interfaces, combo boxes' graphics behavior can change significantly depending on the application being considered. Despite all these variations, almost all combo boxes share a common characteristic, i.e., a rightmost down arrow. Unfortunately, the presence of other GUI elements with down arrows (e.g., vertical scroll bars, custom icons, etc.) could bring to wrong classification results. To properly identify selections corresponding to combo boxes, the GCD module has been programmed to exploit other distinguishing visual details, like the simultaneous highlighting of textual contents and/or other graphics elements. Once a selection has been identified, an algorithm comparable to the one used for dealing with menu items is executed. This algorithm simulates the mouse click over the down arrow, and captures the graphics representation of the newly opened drop-down list. The initial highlighting of the selected item is automatically removed. The mouse is then iteratively moved over the list, the generated highlighting effect is captured, the difference image is computed, and the various regions corresponding to combo box elements are identified (icons and/or text).

- *Text boxes and text areas*: the identification of text boxes and text areas cannot exploit the approach based on highlighting, since these controls do not undergo any particular visible modification. However, graphics interfaces still inform the user about the presence of such elements by modifying the aspect of the mouse pointer into a cursor shape. Thus, during mouse movements, the GDM module monitors the aspect of the mouse

pointer (whose location is known), and when this changes, it is assumed that the mouse is over a text box or text area (until it does not change again). This information is exploited by the GCD to identify the exact location and size of the considered element, and to properly carry out the classification step on it.

5.2 Managing appearing, disappearing and changing widgets

Within graphics interfaces, there could exists widgets that appear, disappear, and change their aspect (from activated to deactivated, from selected to deselected, etc.) when reacting to user interactions. For instance, when working on a word processor, the selection of a block of text usually results in the activation of those buttons enabling cut and paste operations. Similarly, if a document contains portions of text with different alignments, positioning the cursor over a section with a particular alignment results in a variation of the currently selected align button. A more sophisticated example could be represented by the Microsoft Windows Calculator application, where GUI elements can undergo various changes, depending on the particular number representation system being used (i.e., hexadecimal, decimal, octal or binary). Thus, when passing – for instance – from base ten to base two, buttons corresponding to digits from two to nine are automatically disabled; when switching from base two to base sixteen, digits from two to nine are re-enabled, and new buttons corresponding to digits from A to F appear (Figure 5).

Fig. 5. Appearing and disappearing widgets in Microsoft Windows Calculator: (a) interface aspect using a base two number representation system, (b) new buttons enabled working in base sixteen, (c) difference image showing GUI changes, and (d) highlighting effect on a previously disabled button.

To deal with the above issues, the designed virtualization framework continuously moni-

tors application appearance through the GDM module, and automatically identifies dynamic changes possibly occurred in the graphics interface. Each time a new event (generated on the Web OS side) is received, the GDM captures a snapshot of the interface and compares it to the last available GUI representation by computing a difference image. Non-black regions in the difference image actually correspond to dynamic changes. Changes occurred in the regions occupied by the work area(s) are not taken into account, since they are implicitly transmitted to the Web OS interface through continuous graphics-based updates. When a change in the GUI aspect is identified, the reason for change has to be unveiled, in order to let the system act in a proper way. In fact, if the change is due to the appearance of a new element that was not previously displayed, the selection area corresponding to this element has to be computed. Then, the classification step has to be carried out, and the interface description has to be updated with the new information. Otherwise, if the change comes from a previously visible element that is now invisible, a matching between present selection and elements already classified has to be performed. Similar actions need to be carried out for selected, deselected, enabled and disabled elements, which could be now deselected, selected, disabled or enabled, respectively. To identify the reason for change, the GDM automatically drives the mouse pointer over non-black regions, captures a new snapshot of the interface, and computes the difference image with respect to the previously stored graphics representation. When the considered region corresponds to a disabled/invisible element, the highlighting effect is not generated, and no new selection is identified. When the highlighting is generated, a novel selection is identified, and its bounding box is passed to the GCD module for classification and description. In both cases, mouse coordinates are compared with those of previously identified interface elements, whose status is properly updated in the description file.

5.3 Managing the opening of new windows

Another characteristic dynamic behavior of existing interfaces is the opening of new application windows. In the current implementation, the attention has been focused on two types of windows, namely popup and toolbox-like windows (Figures 6 and 7). When a new popup window is opened, it obtains the focus from the system; before being able to come back and work again on the previously active window, the user is asked to carry out some actions on the popup itself (this is the case of common windows like “Save”, “Save as...”, “Print...”, “Preferences”, etc.). On the other hand, toolbox windows do not force the user to immediately work on them; moreover, they do not produce any focus change. Toolbox windows are often used to display previously hidden application features through new GUI elements. The newly opened windows are quite similar to the main application window, as they may contain one or more work areas, and several new graphics elements. Thus, their interfaces can be managed by applying the identification and classification algorithms discussed in the previous section (with the exception of work areas, which require ad hoc rules to be defined in order to be automatically analyzed). However, two novel issues have to be addressed: how to discover the opening of a new window without relying on the events handled by the host operating system, and how to properly distinguish the actual window type.

In the proposed architecture, specific classification rules have been defined to let the GDM module properly execute the above tasks. The management of a popup relies on the fact that the new window, when opened, becomes the current focused element. Focus loss on the main

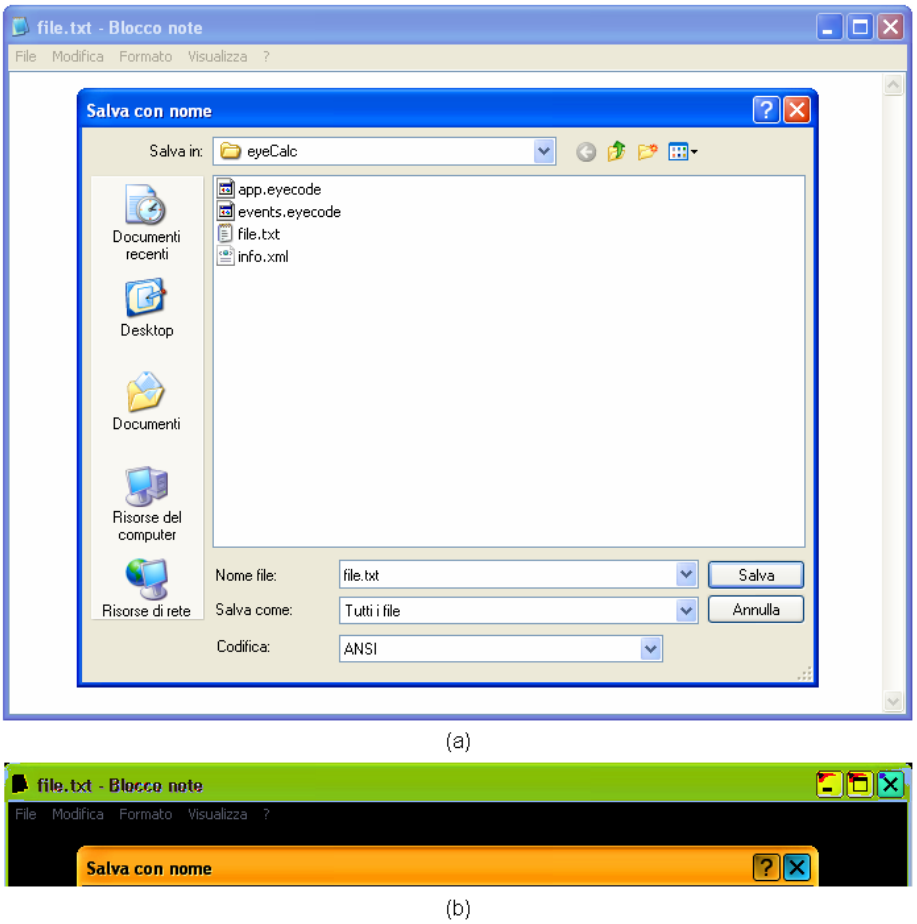

Fig. 6. Dynamic popup window: (a) interface aspect after window opening, and (b) difference image showing main application window's focus loss.

application window's title bar is recorded in the corresponding difference image (Figure 6). When this happens, the GDM module determines the boundaries of the new window by looking for a rectangular region within the difference image. This information is then passed to the GEL and GCD modules, which locate and classify GUI elements by processing them with the same algorithms used on the main application interface. Because of the lack of focus changes, toolbox windows need to be processed in a different way. Like for popups, the GDM locates the boundaries of the newly opened window, and finds out its associated graphics elements. Then, to avoid having the discovered widgets blended in the main GUI, graphics information on toolbox bounding box and/or resizing handles are used to group newly identified elements into a separate container (Figure 7).

Given the fact that the time requested for describing the new window can result in a reduction of the usability degree (since interaction with the interface has to be temporarily inhibited to allow for the execution of the identification and classification steps), the system

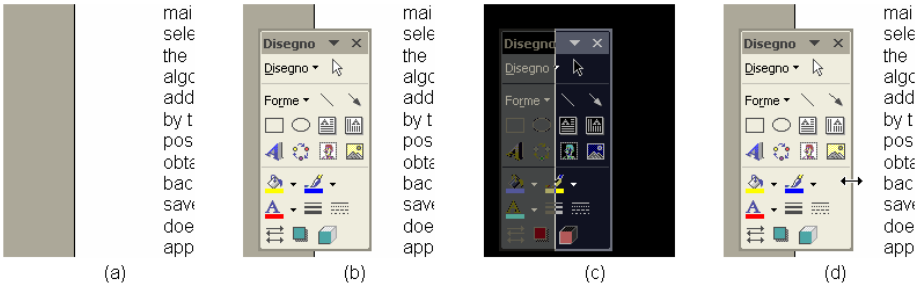

Fig. 7. Dynamic toolbox window: (a) interface aspect before window opening, (b) toolbox window, (c) difference image, and (d) mouse pointer modifications.

has been configured for keeping track of each event causing the opening of a particular window. This may allow to reduce the processing time when the event occurs again. Moreover, in a scenario in which a complete description for all the remote applications is available (generated) before starting up any virtualization session, this particular dynamic behavior can be generally handled a priori, by including dynamically opening windows within the initial interface description file (thus further limiting the transmission of GUI description updates).

6 Implementation Details

The server side of the proposed architecture has been developed as a C++ graphics application under Microsoft Windows and Linux. This application internally implements the logical and functional blocks described in the previous sections. During the system configuration phase, it exploits the GDM functionalities for interacting with the applications to be virtualized, and for producing the initial interface description files. Element identification and classification steps over the selected applications are carried out in an automatic way. Nevertheless, several graphics wizards let the user participate in the above processes, and tune the main configuration parameters. Once the configuration phase is completed, the server application activates the GIM component, that will be in charge of receiving virtualization requests generated by Web OS users, and of managing GUI interactions.

On the client side, a Web OS wrapper application named *eyeVirtualizer* has been implemented using the SDK made available within the eyeOS project [48]. eyeOS is an open source Web OS framework (belonging to the mixed category, according to Section 2), based on a microkernel exporting core services and libraries for simplified Web application development and integration. eyeOS services include a virtual file system, a security and access manager, a process handler, and a graphics subsystem; in turn, system libraries let application developers get access to services and core functionalities like networking and message passing, and allow for the deployment of effective graphics interfaces [49]. In particular, eyeOS GUI toolkit provides a set of basic widgets including windows, icons, labels, buttons, text boxes and text areas, combo boxes, check boxes, tree views, and toolbars. In the eyeOS functional model, Web applications rely on a server-side logic developed in PHP that uses the above services and libraries to carry out its main processing tasks. On the other hand, application interfaces run within a client-side Internet browser, and exploit multiple Web technologies (DHTML, CSS, XML, JavaScript, AJAX, Flash, etc.) to manage the GUI appearance, and to dialogue

with the server-side software.

The integration of the WOV application required to improve native eyeOS functionalities, in order to let the system deal with interface construction, remote interaction, and dynamic updates. For this, the basic eyeOS toolkit was first extended, by designing new widgets matching the XUL's extended lexicon and allowing to construct richer graphics interfaces. Thus, for example, a new menu widget allowing for the development of complex hierarchical structures (including sub menus, icons, and separators) was designed. Moreover, a scroll container capable of hosting other widgets, and allowing for the creation of graphics enabled list boxes was added. Furthermore, the existing combo box widget was modified to create an advanced graphics element mimicking the controls made available by modern GUI toolkits. Then, according to Figure 1, the Web OS application *builder* component was developed, by designing a novel server-side PHP module capable of interacting with the eyeOS libraries to process the XUL description file, reorganize remote interface elements according to locally available widgets, and recreate the virtualized interface. eyeOS widgets signalling capabilities were exploited to construct the *updater* module, and to let it intercept user interaction events to be delivered to the remote virtualization suite. Finally, the Web OS application *manager* was created, and programmed to handle interface customization operations and to deal with the supervising of redrawing of graphics widgets and interface work area(s). In particular, graphics update functionalities were developed by referring to the Jpeg/Tight and Raw/CopyRect encoding capabilities implemented by the TightVNC libraries [21], specifically optimized for effective bandwidth usage. Nevertheless, the above encoding strategies could be possibly replaced by any other solution commonly used in remote visualization architectures [46, 47] without significantly altering the overall philosophy of the considered approach.

As a matter of example, (qualitative) results obtained by virtualizing a desktop instance of Microsoft PowerPoint are illustrated in Figure 8. Here, the interface of the original application was customized to user's taste, and split into multiple windows. Thus, in the main window of the eyeVirtualizer application, some of the menus described in the automatically generated XUL file were removed. The first toolbar was maintained without any change with respect to the desktop version, while elements belonging to the second one were distributed over two different panels. Widgets originally located into separate toolbox windows were aggregated into a single container, and the bottom text area was moved into a new window. Finally, icons corresponding to presentation layout located in right-most area of the interface were reorganized and inserted into a novel scroll container. It is worth observing that, even though in this case customization is based on user's preferences, interface adaptation could also be performed taking into account virtual platform/device requirements.

7 Accuracy of GUI Reconstruction

Web OS application interface completeness (and fidelity), together with interaction latency and bandwidth requirements can be regarded as some of the most important preconditions for the spreading of the proposed virtualization technique. For this, several experimental tests were carried out, with the aim of assessing GUI reconstruction capabilities, as well as of measuring the degree of interactivity of the virtualized application and the associated network overhead. The attention was initially focused on the ability of the server-side architecture to recognize (i.e., localize and classify) interface elements. For this, a common window-based

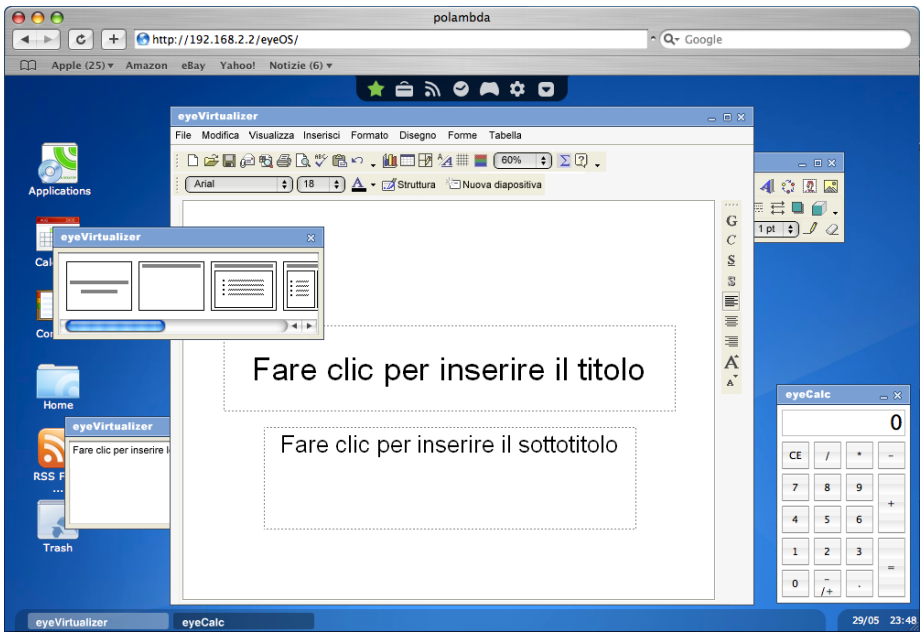

Fig. 8. Microsoft PowerPoint interface virtualized in eyeOS (under Safari on MacOS X).

operating system was selected (namely, Windows XP), and a set of applications commonly used on desktop machines and belonging to several representative categories was identified as test bed. Application set included Apple Safari 4 and Mozilla Firefox 3.06 (Web browsing), Adobe Photoshop CS3 and GIMP 2.6.5 (graphics editing), Microsoft Office XP PowerPoint (presentation), OpenOffice 3.0.1 Calc (spreadsheet), Microsoft WordPad (word processing), Microsoft Calculator (calculation), and Blender 248.1 (3D modeling).

At first, the GUIs of the selected applications were automatically analyzed (soon after application startup, i.e., without any user customization), by generating corresponding XUL-based descriptions reporting the outcomes of localization and classification steps. In order to investigate the advantages of the considered approach with respect to alternative graphics-unaware solutions mentioned in Section 4, results above were compared with those obtained by gathering accessibility/automation features exposed by selected applications. For this, the Microsoft UISpy tool was used [50]. Finally, in order to compare the above findings with a reliable ground-truth, elements constituting the interfaces of the above applications were manually identified and categorized. Comparison did not intentionally encompass other approaches solely based on image-processing techniques (like, for instance, those presented in [31, 32, 33]). This choice was motivated by the fact that, when these approaches are used, accuracy performances strongly depend on the particular graphics style being considered in designing the classification algorithm; moreover, because of the introduction of a GUI-supported localization step, it is reasonable to expect the designed two-step methodology to achieve better results, supposed that the same classification rules are used.

Detailed localization and classification results obtained by manual inspection, with the

UISpy tool, and with the proposed virtualization framework on a (representative) subset of the considered applications are reported in Table 2. Aggregate results (normalized with respect to the manual ground-truth) for the whole application set are plotted in Figure 9.

With respect to the results obtained by using the UISpy tool, the considered applications can be roughly classified in three main categories, depending on their accessibility/automation attitudes.

- The first category groups applications strongly oriented to accessibility/automation (e.g., Calculator, WordPad, PowerPoint), whose GUI organization in terms of widgets location and behavior can be completely identified. Applications in this category can be developed, for instance, by using the Microsoft Win32, Windows Forms, and Microsoft Windows Presentation Foundation (WPF) frameworks; detailed characteristics of possibly existing custom elements can be exposed via the Microsoft UI Automation API. Localization and classification results for the WordPad application can be found in Table 2.a.

- The second category assembles applications that expose, often in a partial way, information regarding graphics element location, but that only provide a few indications concerning the specific widget type; thus, in most cases, it is not possible to distinguish among menus, buttons, combo boxes, etc. Applications in this category can be either developed with non native graphics libraries (e.g., QT, for Safari, or GDK/GTK, for GIMP) not providing a complete integration with the UI Automation API, or by relying on custom elements derived from a standard type, e.g., a panel, but not implementing the requested accessibility/automation interface (like Photoshop). Results obtained with Photoshop and GIMP are reported in Tables 2.b and 2.c, respectively.

- Finally, the third category groups applications that are mostly accessibility/automation-unaware. When launched on applications belonging to this latter class, the UISpy tool was able to localize and classify less than three percent of the available interface elements. Applications in this category are usually developed by exploiting completely customized widgets (e.g., Firefox), or by using graphics toolkits not supporting Microsoft UI accessibility/automation (like OpenGL for Blender, Java Swing for OpenOffice, etc.). Localization and classification results achieved on the Firefox GUI are illustrated in Table 2.d.

In summary, results in Table 2 confirmed that there exist frequently used applications whose interface cannot be analyzed and described by means of approaches solely based on programmatic or reverse engineering-oriented techniques. On the contrary, the above results provided promising indications concerning the applicability of the designed image-based techniques, which represent the basis of the proposed virtualization approach. In fact, being based on graphics behaviors that are common – possibly with some changes – to the all the considered applications (as well as to the majority of graphics applications running on window-based operating systems), the designed recognition algorithms proved to work well in all the experimental scenarios.

More precisely, by comparing aggregate results (in Figure 9) achieved by running the localization algorithm soon after application startup, as well as after interface dynamic updates, it can be observed that in the first step, the enabled elements (on which highlighting

Table 2. A comparison of localization and classification accuracy: results obtained by manual inspection, by the Web OS-based approach (at application startup and after user interaction), and by the UISpy tool

(a) Microsoft WordPad

Element	**Manual**	**Web OS-based framework**			**UISpy tool**	
		Loc. (startup)	**Loc. (upd.)**	**Class.**	**Loc.**	**Class.**
Combo box	3	3	3	3	3	3
Menu	6	6	6	6	6	6
Menu item	39	39	39	39	39	39
Button	22	18	22	22	22	22
Text field	4	4	4	4	4	4
Total	**74**	**70**	**74**	**74**	**74**	**74**

(b) Adobe Photoshop CS3

Element	**Manual**	**Web OS-based framework**			**UISpy tool**	
		Loc. (startup)	**Loc. (upd.)**	**Class.**	**Loc.**	**Class.**
Combo box	4	1	4	4	2	2
Menu	11	10	11	11	10	10
Menu item	223	209	223	223	209	209
Button	71	60	66	66	7	7
Text field	10	6	10	10	10	10
Slider	4	4	4	4	0	0
Scroll bar	3	3	3	3	3	3
Total	**326**	**293**	**321**	**321**	**241**	**241**

(c) GIMP 2.6.5

Element	**Manual**	**Web OS-based framework**			**UISpy tool**	
		Loc. (startup)	**Loc. (upd.)**	**Class.**	**Loc.**	**Class.**
Combo box	3	3	3	3	3	0
Menu	12	11	12	12	12	0
Menu item	173	152	173	173	173	0
Button	52	46	52	52	52	4
Text field	4	4	4	4	4	0
Slider	2	2	2	2	2	0
Check box	4	4	4	4	4	0
Scroll bar	2	2	2	2	2	0
Total	**252**	**224**	**252**	**252**	**252**	**4**

(d) Mozilla Firefox 3.0.6

Element	**Manual**	**Web OS-based framework**			**UISpy tool**	
		Loc. (startup)	**Loc. (upd.)**	**Class.**	**Loc.**	**Class.**
Combo box	1	1	1	1	0	0
Menu	9	8	8	8	0	0
Menu item	61	61	61	61	0	0
Button	11	10	11	11	3	3
Text field	2	2	2	2	0	0
Total	**84**	**82**	**83**	**83**	**3**	**3**

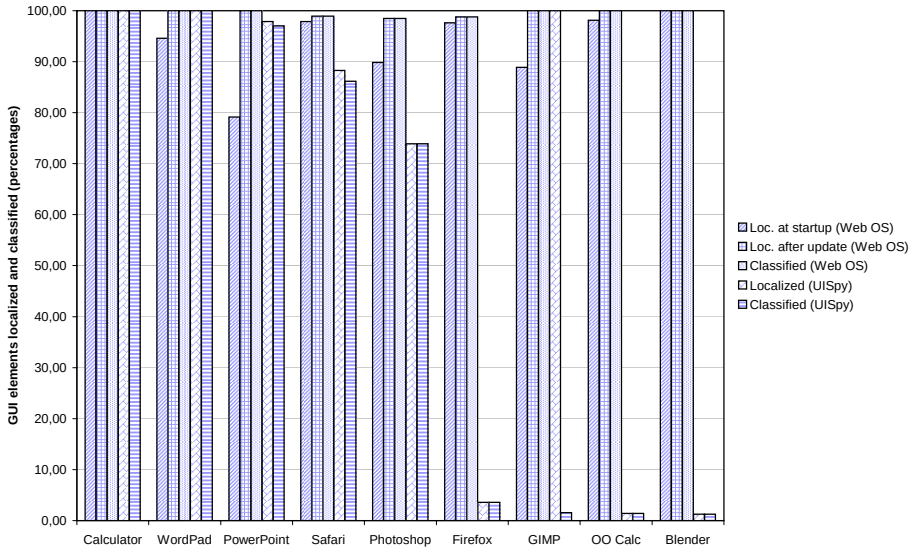

Fig. 9. A comparison between localization and classification results obtained by means of the designed Web OS-based virtualization approach and by exploiting accessibility/automation information gathered through the UISpy tool (percentages normalized with respect to manual inspection ground-truth).

and other visual modifications occurred) were correctly identified by the GEL module. In the next localization steps activated by the GDM in response to user interactions, almost all the remaining elements were successfully processed. Moreover, when classification accuracy is considered, the designed solution still outperforms results that can be achieved by resorting to accessibility/automation information; in fact, being tailored to the specific graphics elements composing the selected applications, the experimented classification algorithm proved to be capable of correctly categorizing almost all the localized interface elements. It is worth remarking that in the present implementation, not all the possible graphics "schemes" were considered; thus, other applications characterized by strongly different interface elements would require the definition of new rules and could result in different classification results. However, given the modular approach adopted in the definition of the classification technique and the possibility of reusing the same (or comparable) rules for multiple environments and applications, significant time and cost savings can be expected, expecially when comparing the effort associated with the definition of image processing-based rules against the effort required for porting a number of different desktop applications on multiple platforms.

8 Network Overhead and Interactivity

Although the focus of this work is on the possibility of building up a unified desktop environment by migrating existing applications to Web OSs, as anticipated in Section 3 the separation between GUI and work area(s) that can be achieved through the considered virtualization approach can give additional advantages possibly encompassing bandwidth occupation and interaction latency. With the aim of analyzing system performances from the above points of view, several reference scenarios corresponding to different types of interaction with a remote

application were identified.

(a) User directly interacts with the work area(s), without introducing any change in the GUI: this could be the case, for instance, of a user working with a calculator application, and inputting numbers by means of his/her keyboard. In this case, virtualization approaches like Remote Desktop and VNC would result in the transmission of frame buffer blocks recording graphics changes occurred in the numeric work area. In turn, in X Window-based approaches the synchronization between client and server would be achieved through the transmission of low-level drawing primitives (generally including also raw image data).

(b) User works on the application by using its GUI (without requiring any dynamic update to the interface) and expects to appreciate interaction results through work area updates: by referring again to the above situation, this could be the case of a user operating on the calculator application by using interface elements (e.g., number and operation buttons). In this case, traditional remote visualization approaches would deliver screen updates (either as frame buffer blocks or graphics primitives) recording changes in both work area and GUI elements.

(c) User carries out GUI interactions (not requiring any dynamic update to the interface) without explicitly requesting any work area update: this could be the case of a user that operates on a graphics interface e.g., by opening, inspecting and scrolling menus and combo boxes without activating them (otherwise this interaction would fall into the previous category). In this case, user interaction generally produces changes in the interface elements (e.g., highlighting) and may also result in GUI and work area occlusions (e.g., due to a combo box list overlapping the underlying graphics area). To manage this kind of interaction, virtualization approaches like Remote Desktop and VNC would deliver updates recording both GUI and work area changes resulting from widget changes as well as from the appearance and disappearance of the occluding graphics element. On the other hand, X Window-based approaches would recognize the fact that an occluded area has become visible again, thus reducing the amount of information exchanged; nevertheless, they would still send graphics updates for managing the remaining changes.

(d) User carries out GUI interactions introducing dynamic changes in the interface: this could be the case of a user selecting the "Save as..." menu item in a given application. This operation would generally result in the opening of a new interface window. In this case, traditional virtualization approaches would act as in the previous scenario; moreover, with respect to the above example, consecutive selections of the same menu item would be managed by delivering again complete graphics updates.

By considering the above scenarios, three main issues characterizing traditional virtualization approaches can be identified:

- delivery of "aesthetics"-only GUI updates (due, for example, to graphics changes used for attracting user's attention);
- delivery of updates corresponding to unwanted changes to the work area (due, for instance, to overlapping);

- delivery of repeated updates corresponding to the same (dynamic) modification in the interface.

Thus, besides using available bandwidth to deliver work area updates, in some cases traditional virtualization approaches use (and possibly waste) network resources to deliver contents that were not explicitly requested by the user, but that are "only" needed to maintain client-server graphics synchronization. In some cases, the above behavior can have a dramatic impact on interaction delay, especially when low-bandwidth and/or high latency connections are considered.

The Web OS-based virtualization approach discussed in this paper implicitly allows to cope with the above limitations. In fact, in the devised solution, only necessary work area changes are delivered through graphics updates. Thus, with respect to the above issues, aesthetics GUI changes can be recreated directly on the client side, without any update transmission; similarly, server-side changes due to overlapping do not have to be delivered to the client viewer application; finally, dynamic changes to the interface are managed by computing and transmitting the updated GUI description only the first time changes occur, whereas following modifications can often refer to the previously available description. In this way, a significant reduction of requested network traffic can be expected, with a consequent reduction of the overall interaction latency.

In order to measure bandwidth saving and interaction speedup that can be possibly ensured by the considered architecture, several experimental tests were carried out by making reference to the scenarios (a)–(d) of the above classification. With the aim of providing a comprehensive evaluation, results obtained by the proposed Web OS-based virtualization approach were compared with those achieved in VNC and X Window oriented scenarios. In particular, the Java-based TightVNC client [21] and the WeirdX X Window server [22] were used; this choice allowed to carry out a fair analysis, since both the solutions are based on the Web (that represents the actual focus of the proposed virtualization technique) and can be configured to deliver graphics updates by exploiting comparable graphics encodings (as discussed in Section 6).

8.1 Bandwidth usage

The evaluation of bandwidth usage was carried out by first comparing the proposed Web OS-based virtualization approach with a VNC-oriented solution under Microsoft Windows. In particular, for investigating scenario (a), the Calculator application was used to perform some computations by exploiting the keyboard as input device (i.e., no change was introduced in the application interface). To analyze scenario (b), the same computations were executed by resorting only to GUI buttons. For scenario (c), the Blender application was remotely controlled by solely interacting with menus and combo boxes. In particular, a number of menus and combo boxes were opened, inspected, and closed consecutively, without selecting any item; thus, interaction did not result into any visible change to the work area. Lastly, scenario (d) was simulated by experimenting a more complex interaction based on the WordPad application. In this case, menu-based interactions introduced heavy changes to both the interface and the work area (e.g., due to dynamic updates associated with the menu-initiated

Table 3. Bandwidth requirements for Web OS and VNC-based virtualization approaches in the selected interaction scenarios.

	Calculator (keyboard) Scenario (a)		**Calculator (GUI)** Scenario (b)		**Blender** Scenario (c)		**WordPad** Scenario (d)	
	VNC	Web OS	VNC	Web OS	VNC	Web OS	VNC	Web OS
Virtual. area (width)	260	253	260	253	796	796	796	796
Virtual. area (height)	260	23	260	23	564	348	564	438
Graphics blocks	44	41	280	41	103	0	569	22
GUI descriptions	0	1	0	1	0	1	0	5
KB transm. (before int.)	16	18	16	18	28	72	40	39
KB transm. (during int.)	2	2	33	2	368	0	519	36
KB transm. (total)	18	20	49	20	396	72	559	75

"Save as...", and "Find and Replace" operations).[a]

Experiments above were carried out over a local network connection. The TightVNC server was set up for virtualizing the whole application GUI, with a 100 ms update polling cycle. Both the frameworks were configured for transmitting graphics changes using either Jpeg or Tight encoding with maximum compression and best image quality. Average results obtained in the above scenarios are illustrated in Table 3, where rows tabulate the size of the application area being virtualized (width and height), the number of updated graphics blocks (Jpeg/Tight) and GUI descriptions transmitted during the experiment, the amount of data (in KBytes) delivered by the server *before* any interaction (corresponding to the transmission of either the first image update or the GUI description), and the amount of data sent to the virtualized application *during* interaction. Finally, the total amount of data exchanged in the considered scenario is reported in the last row.

By analyzing Table 3, it can be observed that when interaction does not produce any modification to the GUI, as in scenario (a), the amount of data exchanged with the VNC and the Web OS approaches is almost comparable. However, it is worth considering that in the VNC-based scenario, data exchange before interaction is due to the transmission of the initial graphics representation of the application interface. On the other hand, in the Web OS solution the byte count corresponds to the graphics delivery of the work area (whose reduced size is 253×23 pixels), of the interface description, and of the linked graphics resources.

In scenario (b), GUI manipulations result into both interface and work area updates. Thus, the VNC-based implementation requires a significantly larger data exchange; moreover, interaction results into a definitely higher number of graphics updates (related in some way to the overall interaction latency, as discussed in Section 8.2). On the other hand, the designed Web OS virtualization approach does not introduce any additional overhead with respect to scenario (a).

Scenario (c) is characterized by many interactions strongly altering the aspect of the application interface (because of the opening and closing menus periodically hiding both GUI and work area). In this case, the transmission of the application description to the Web OS application requires significantly more bandwidth than the delivery of the VNC startup image. Nevertheless, with the Web OS approach, GUI interactions can be recreated locally

[a]Videos captured on the server-side during remote interaction are available on-line at the following address: http://130.192.5.71:8080/webos/vnc/bandwidth/

on the client side, without requiring any update transmission; on the other hand, with VNC, each GUI update requires to deliver graphics information corresponding to a screen area than can be significantly large and rich of contents. Thus, in the considered experiment, VNC requires roughly five times more bandwidth than the Web OS approach.

Lastly, in scenario (d) visual changes that are mainly due to GUI dynamics updates are considered. Data transmitted before starting the interaction are roughly the same. In fact, despite the reduced size of the work area, in the Web OS scenario GUI description has to be initially delivered; however, during interaction, the transmission of work area and GUI description updates requires definitely fewer bandwidth. In fact, besides requiring reduced network resources for handling work area updates, in this case the designed Web OS implementation benefits from the possibility of reusing previously transmitted GUI descriptions to manage known interface changes (i.e., related to previously opened windows).

A similar analysis was then carried out by comparing the proposed virtualization approach with a Java-based X Window framework under Linux. Scenarios (a) and (b) were evaluated again by performing some computations with a calculator tool; in this case, the gcalctool application of the Gnome environment was used. To analyze scenario (c), the Mozilla Firefox application was considered; several menus and combo boxes were opened, inspected and closed without altering the underlying work area. Finally, to evaluate scenario (d), the GIMP graphics tool was selected and a number of interaction commands altering both the GUI and the work area were issued. Specifically, during the interaction a recently saved 400×300 pixels image was opened; then, the color balance tool was used to adjust the color distribution and the "Save as..." menu item was activated to save the image over the original file; finally, the image was scaled down to 300×225 pixels and saved again over the previous file (in both cases, a window allowing to select the destination filename was opened, and a further confirmation was requested to overwrite the existing file).[b]

In order to perform a fair comparison, the same network configuration was used, and image compression was disabled in both the frameworks. Moreover, data exchanges corresponding to the opening of splash screen possibly displayed by some applications before startup (e.g., by gcalctool and GIMP) were discarded. Average results obtained in the considered scenarios are reported in Table 4. For the X Window system, the number of graphics blocks corresponds to the number of putImage requests issued by the X client and resulting in the transmission of raw graphics data.

As with the VNC technique, when interaction effects do not involve the GUI, like in scenario (a), bandwidth used by the X Window and Web OS virtualization approaches is almost comparable; differences are due to the different mechanism used for creating (i.e., drawing or building, respectively) the application interface before interaction and by the slightly different coding scheme and header information used for transmitting raw graphics blocks.

However, in scenario (b) GUI drawing operations performed by the X Window system result in a higher bandwidth usage, whereas the Web OS virtualization approach uses exactly the same bandwidth of scenario (a). This is due to the fact that, even though X Window is somehow aware of the separation between GUI and work area, it treats update operations in

[b] Videos captured on the server-side during remote interaction are available on-line at the following address: http://130.192.5.71:8080/webos/x/bandwidth/

Table 4. Bandwidth requirements for Web OS and X Window-based virtualization approaches in the selected interaction scenarios

	gcalctool (keyboard)		**gcalctool (GUI)**		**Firefox**		**GIMP**	
	Scenario (a)		Scenario (b)		Scenario (c)		Scenario (d)	
	X	Web OS	X	Web OS	X	Web OS	X	Web OS
Virtual. area (width)	295	273	295	273	806	800	900	684
Virtual. area (height)	264	70	264	70	626	452	582	498
Graphics blocks	90	76	264	76	416	0	1174	118
GUI descriptions	0	1	0	1	0	1	0	5
KB transm. (before int.)	31	35	31	35	504	575	818	951
KB transm. (during int.)	232	239	404	239	849	0	2911	1479
KB transm. (total)	263	274	435	274	1353	575	3729	2430

a similar way, by transmitting suitable drawing primitives (in this case required to update the work area and to handle aesthetics changes to the GUI).

The above behaviour is even more evident in scenario (c), where updates are solely related to changes in GUI appearance. In this case, even though the interface drawing step in the X Window system requires the exchange of a lower amount of data, during interaction the Web OS virtualization approach does not require any significant data transfer. On the other hand, in order to properly manage interaction, the X Window system results in the transmission of a significantly higher number of graphics blocks.

In a scenario (d), a more graphics intensive application is considered. As it can be seen, during interaction both the virtualization approaches exchange of a larger amount of data with respect to previous scenarios. This is due to the fact that both the GUI and the work area need to be refreshed several times. However, the Web OS virtualization approach results in a lower bandwidth usage, as data used for constructing the various windows opened need to be delivered only once, whereas with the X Window system drawing requests and associated graphics blocks are sent by the X client at each iteration.

In summary, experimental tests showed that, in typical interaction scenarios where GUI updates are not negligible, Web OS virtualization is capable of allowing for a saving of network resources with respect to both VNC and X Window-based techniques, which could be possibly associated with a reduction of the overall interaction latency.

8.2 Interaction latency

In the considered virtualization scenarios, the degree of interactivity with the virtualized application is strongly affected by the latency experienced by the user during remote manipulation. Latency can be described as the time required for updating client application interface and/or work area(s) by either delivering graphics updates, or by locally recreating GUI modifications using widgets available in the virtualization environment (possibly requesting updated GUI descriptions). Depending on the particular virtualization strategy being considered, latency can be influenced by different factors, including network capabilities and conditions, interface size and complexity, interaction type, etc. In order to better analyze the contributions of factors above, the designed Web OS-based framework was compared to the considered virtualization solutions by evaluating performances in the selected reference scenarios under different network conditions.

Table 5. Mean interaction latency (in milliseconds) in Web OS and VNC-based virtualization scenarios in a WLAN configuration

	WLAN mean round-trip time (ms)	3	
	WLAN average bandwidth (Mbps)	3.5	
		VNC	Web OS
Calculator Scenario (a)	No. of updates (work area/GUI)	1/0	1/0
	Update (work area): delay	11	13
Calculator Scenario (b)	No. of updates (work area/GUI)	1/2	1/0
	First update (button down): delay	9	-
	Second update (work area): delay	165	167
	Third update (button up): delay	186	-
Safari Scenario (c)	No. of updates (work area/GUI)	1/1	0/0
	First update (menu opened): KB transm.	1	0
	First update (menu opened): delay	30	0
	Second update (menu closed): KB transm.	44	0
	Second update (menu closed): delay	72	0
Safari Scenario (d)	No. of updates (work area/GUI)	2/2	0/1
	Upd. interface description size	-	18
	Gen. of GUI desc. after dynamic upd.: delay	-	2124
	First update (window opened): KB transm.	103	0
	First update (window opened): delay	265	2174
	Second update (window closed): KB transm.	340	0
	Second update (window closed): delay	792	0
	First interaction: overall delay	1057	2174
	Third update (window opened again): delay	265	17
	Fourth update (window closed again): delay	792	0
	Second interaction: overall delay	1057	17
	Overall delay (both interactions)	2114	2191

In particular, the proposed virtualization method was first compared with the selected Java-based VNC client under Microsoft Windows. For simulating scenario (a), the Calculator application was remotely controlled using both the Web OS-based framework and the TightVNC Web client. Latency was measured during consecutive presses of the same numeric key. Scenario (b) was analyzed by interacting again with the Calculator application; however, in this case, one of the numeric buttons of the GUI was used. To experiment scenario (c), the Safari application was selected; interaction was achieved by opening and closing the "File" menu (thus occluding a portion of the work area). Finally, scenario (d) was simulated by performing two times the following operation: selection of the "Open File..." menu item of the Safari Web browser, and cancelation of the operation (by closing the newly opened window).[c]

Average results obtained on WLAN and ADSL network connections for scenarios (a)–(d) are reported in Tables 5 and 6, respectively. Depending on the scenario being considered, results are tabulated on a different number of rows; in fact, for each scenario, relevant contributions to the overall interaction latency are separately considered. It is worth observing that interaction latency was measured once the virtualized application representation was available at the client side (either through VNC-based graphics updates or GUI description transmission).

¿From Tables 5 and 6 it can be easily observed that, in scenario (a), latency can be regarded as the time passing between a given key press and the corresponding work area

[c] Videos captured on the server-side during remote interaction are available on-line at the following address: http://130.192.5.71:8080/webos/vnc/latency/

Table 6. Mean interaction latency (in milliseconds) in Web OS and VNC-based virtualization scenarios in a ADSL configuration

	ADSL mean round-trip time (ms)	58	
	ADSL average bandwidth (Mbps)	1.2	
		VNC	Web OS
Calculator Scenario (a)	No. of updates (work area/GUI)	1/0	1/0
	Update (work area): delay	67	70
Calculator Scenario (b)	No. of updates (work area/GUI)	1/2	1/0
	First update (button down): delay	66	-
	Second update (work area): delay	224	229
	Third update (button up): delay	247	-
Safari Scenario (c)	No. of updates (work area/GUI)	1/1	0/0
	First update (menu opened): KB transm.	1	0
	First update (menu opened): delay	95	0
	Second update (menu closed): KB transm.	44	0
	Second update (menu closed): delay	383	0
Safari Scenario (d)	No. of updates (work area/GUI)	2/2	0/1
	Upd. interface description size	-	18
	Gen. of GUI desc. after dynamic upd.: delay	-	2124
	First update (window opened): KB transm.	103	0
	First update (window opened): delay	724	2327
	Second update (window closed): KB transm.	340	0
	Second update (window closed): delay	2293	0
	First interaction: overall delay	3017	2327
	Third update (window opened again): delay	724	79
	Fourth update (window closed again): delay	2293	0
	Second interaction: overall delay	3017	79
	Overall delay (both interactions)	6034	2406

update in the client application. In this case, application GUI does not undergo any change, and latency strongly depends on network round-trip time; this is due to the fact that work area updates result in the transmission of an extremely limited amount of data (see Table 3), that can be easily managed by all the considered network connections. Slight differences between the VNC technique and the Web OS-based approach can be due to the overhead associated to event processing within the eyeOS framework, and to the increased complexity of the server-side architecture (that is programmed to continuously check for dynamic changes in the interface).

In scenario (b), where interaction is controlled by an interface button, the VNC-based remote framework delivers changes corresponding to both work area and updated button status (i.e., button highlighted, button down, and button up). On the other hand, Web OS only manages changes affecting the work area. Thus, with the Web OS-based implementation latency is slightly lower than with VNC, as interaction can be considered as concluded as soon as expected work area update has been received (while in the VNC case, an additional update is still requested in order to restore the original interface appearance).

Things are dramatically different when scenario (c) is considered: in this case, interaction results in the transmission of two graphics updates, the first one corresponding to the newly opened menu, and the second one corresponding to the restored graphics space previously hidden by the menu. Given the fact that the first update require to transmit approximately 1 KB of data, latency in VNC is mainly related to network round-trip time. On the other hand, the second update results in the transmission of 44 KB of data: in this case, latency

is strongly influenced by the transmission delay, especially when a low bandwidth connection is considered. By contrast, when considering the Web OS approach, it can be observed that interface updates are completed managed by the local application, without introducing any latency overhead.

Finally, in scenario (d) GUI dynamic updates are taken into account: in this case, interaction initially results in the opening of a new window. Thus, the Web OS approach requires to analyze the GUI of the newly displayed interface; then, the generated description has to be delivered to the client side. On a local network, these steps result in a higher latency than with the VNC-based approach (where latency is determined by the transmission of a graphics update of roughly one hundred KB). However, closing the window does not introduce any latency in the Web OS scenario; on the other hand, the VNC-based solution is characterized by a latency that is even higher than before, because the amount of data that has to be delivered is significantly higher. In summary, when the aggregated latency for the two interactions is considered (i.e., opening and closing of the window), starting from an ADSL network connection the Web OS solution is capable of achieving better performances. Moreover, the experimental test selected for scenario (d) also allowed to analyze Web OS performances for repeated (non necessarily consecutive) interactions with the same GUI element, which represent a common usage pattern. In this case, it can be easily observed that, with the proposed approach, the latency experienced for the second interaction is almost close to the round-trip time; this is due to the fact that previously available description is used, when possible. On the other hand, VNC does not distinguish between repeated interactions on the same interface element, and retransmits both graphics updates (thus introducing the same latency experienced in the first interaction).

Experiments on interaction latency were then repeated by comparing the Web OS and the X Window-based virtualization approaches. In order to obtain data to be possibly compared with results obtained using VNC virtualization, scenarios similar to those exploited in the previous analysis were considered; nonetheless, given the fact that the use of X-Window virtualization is limited, on the server side, to Unix-like environments, other applications running under a different operating system were used. In particular, for scenarios (a) and (b), the Calculator tool was replaced by the Gnome gcalctool application, and the same tests performed with VNC were carried out. To experiment scenarios (c) and (d), a more graphics intensive application was used. Specifically, in scenario (c) the GIMP application was run, and interaction latency was measured during the opening and closing of a menu, as well as during a scroll operation on the patterns window. Finally, to evaluate scenario (d) a 400×300 pixels image was loaded in the GIMP application, and the toolbox window for controlling brightness and contrast was opened and closed, at first; then, the window was opened again and an existing preset was selected from a list of previously used ones; finally, selected preset was applied to the opened image thus varying its brightness.[d]

Average results experienced in the above scenarios on WLAN and ADSL network connections are illustrated in Tables 7 and 8, respectively. As in Tables 5 and 6, the various scenarios are analyzed by resorting to a variable number of rows, in order to precisely identify the aspects determining the overall delay. With respect to the previous analysis, specific indi-

[d]Videos captured on the server-side during remote interaction are available on-line at the following address: http://130.192.5.71:8080/webos/x/latency/

Table 7. Mean interaction latency (in milliseconds) in Web OS and X Window-based virtualization scenarios in a WLAN configuration

	WLAN mean round-trip time (ms)	3	
	WLAN average bandwidth (Mbps)	3.5	
		X	Web OS
gcalctool	No. of updates (work area/GUI)	1/0	1/0
Scenario (a)	Update (work area): delay	12	15
gcalctool	No. of updates (work area/GUI)	1/2	1/0
Scenario (b)	First update (button down): delay	80	-
	Second update (work area): delay	158	182
	Third update (button up): delay	195	-
GIMP	No. of updates (work area/GUI)	0/2	0/0
Scenario (c)	First update (menu opened): KB tr./KB graph. bl.	102/84	0/0
	First upd. (menu opened): prim./graph. bl.	756/112	-/0
	First upd. (menu opened): delay	336	0/-
	Third update (scrolling): KB tr./KB graph. bl.	263/244	0/0
	Third upd. (scrolling): prim./graph. bl.	430/86	-/0
	Third upd. (scrolling): delay	276	0
GIMP	No. of updates (work area/GUI)	1/4	1/1
Scenario (d)	Upd. interface description size	-	75
	Gen. of GUI desc. after dynamic upd.: delay	-	1185
	First update (win. opened): KB tr./KB graph. bl.	119/78	0/0
	First upd. (win. opened): prim./graph. bl.	921/65	-/0
	First upd. (win. opened): delay	422	1220
	Second update (win. closed): KB tr./KB graph. bl.	23/12	0/0
	Second upd. (win. closed): prim./graph. bl.	143/7	-/0
	Second upd. (win. closed): delay	68	0
	First interaction: overall delay	490	1220
	Third update (win. opened): KB tr./KB graph. bl.	119/78	0/0
	Third upd. (win. opened): prim./graph. bl.	921/65	-/0
	Third upd. (win. opened): delay	422	12
	Fourth update (list op.): KB tr./KB graph. bl.	44/23	0/0
	Fourth upd. (list op.): prim./graph. bl.	450/26	-/0
	Fourth upd. (list op.): delay	191	0
	Second interaction: overall delay	613	12
	Fifth update (selection): KB tr./KB graph. bl.	15/8	0/0
	Fifth upd. (selection): prim./graph. bl.	347/24	-/0
	Fifth upd. (selection): delay	141	0
	Sixth update (apply preset): KB tr./KB graph. bl.	523/489	503/503
	Sixth upd. (apply preset): prim./graph. bl.	998/47	-/26
	Sixth upd. (apply preset): delay	599	238
	Third interaction: overall delay	740	238
	Overall delay (all the interactions)	1843	1470

cations concerning the number of drawing primitives (and graphics blocks) transferred on the network during interaction, as well as the amount of KBytes corresponding to image-based graphics updates (with respect to the overall network load) have been added, as they have a strong influence on the overall interaction latency and allow to point out distinguishing features of the proposed approach with respect to X Window-based techniques.

As expected, results obtained with the Java-based X Window server in scenario (a) are almost comparable to that achieved with the Web OS-based virtualization approach (and VNC, as well), as in this case only (small) work area updates are transmitted with a comparable coding scheme. The slightly higher latency with respect to VNC is due to the fact that uncompressed data corresponding to a larger interaction area are transmitted, in this case.

As already experienced with the VNC, in scenario (b) the proposed Web OS-based approach achieves better performances with respect to the X Window system. This is due to the fact that, similarly to the VNC, the X Window system is not able to locally cope with graphics changes affecting the GUI. In fact, in order to properly manage the interaction, it is required to transmit graphics primitives to both handle work area and button redrawing, whereas the Web OS virtualization approach only has to deal with work area updating.

In scenario (c), the interaction with the X Window system results in the transmission of three updates, basically needed to draw the opened/closed menu and the scrolled patterns window area. However, it is worth observing that, differently than with the VNC virtualization approach, in order to manage the menu closing, the X Window system does not require any significant data transfer (hence, rows regarding this step are omitted). As it can be seen from Tables 7 and 8, with X Window the interaction latency is related to the number and type of the particular drawing primitives transmitted from the X client to the X server, as well as to the bandwidth and round-trip time in the selected network configuration. On the contrary, with the proposed Web OS virtualization approach, all the updates are handled locally, without introducing any interaction latency.

Lastly, in scenario (d) three macro-interactions can be identified. During the first interaction, a new window is opened and closed. The opening of the new window requires the Web OS virtualization framework to analyze its new GUI and to transmit a newly generated interface description. For this interaction, the above step makes the Web OS based solution slower than the X Window virtualization approach. However, during the second macro-interaction, the window is opened again and its widgets are manipulated. In this case, the delay introduced by the Web OS virtualization approach roughly corresponds to the round-trip time as the previously transmitted GUI description is used; moreover, the opening of the combo box containing presets is handled without any interaction with the remote side (as presets information had been already transmitted with the window description). Thus, in the proposed approach, the second interaction is carried out with a negligible latency overhead. On the other hand, the X Window system transmits both the graphics primitives required to manage the drawing of the window area as well as of the presets list; in this case, a latency that is even higher than the one experienced in the first interaction is introduced. Finally, in the third macro-interaction both the GUI and the work area need to be updated (a preset is selected and applied to the image). Both the virtualization approaches need to transmit a significant amount of data corresponding to raw image updates used to refresh the work area. However, the high number of primitives sent by the X client makes the X Window system significantly slower than the proposed approach (this behavior is also due to the larger amount of data exchanged in the uplink direction to maintain synchronization). When the overall delay corresponding to the three interactions is considered, it can be observed that the proposed approach allows achieving a lower interaction latency in both the network scenarios considered (with a more evident speedup on the lower-bandwidth asymmetric ADSL link with higher round-trip times).

In summary, experimental tests demonstrated that in almost all the considered scenarios, the Web OS virtualization approach is capable of reducing the overall interaction latency, thus possibly improving remote application usability and interactivity.

Table 8. Mean interaction latency (in milliseconds) in Web OS and X Window-based virtualization scenarios in a ADSL configuration

	ADSL mean round-trip time (ms)	58	
	ADSL average bandwidth (Mbps)	1.2	
		X	Web OS
gcalctool	No. of updates (work area/GUI)	1/0	1/0
Scenario (a)	Update (work area): delay	78	83
gcalctool	No. of updates (work area/GUI)	1/2	1/0
Scenario (b)	First update (button down): delay	80	-
	Second update (work area): delay	237	245
	Third update (button up): delay	305	-
GIMP	No. of updates (work area/GUI)	0/2	0/0
Scenario (c)	First update (menu opened): KB tr./KB graph. bl.	102/84	0/0
	First upd. (menu opened): prim./graph. bl.	756/112	-/0
	First upd. (menu opened): delay	778	0/-
	Third update (scrolling): KB tr./KB graph. bl.	263/244	0/0
	Third upd. (scrolling): prim./graph. bl.	430/86	-/0
	Third upd. (scrolling): delay	699	0
GIMP	No. of updates (work area/GUI)	1/4	1/1
Scenario (d)	Upd. interface description size	-	75
	Gen. of GUI desc. after dynamic upd.: delay	-	1185
	First update (win. opened): KB tr./KB graph. bl.	119/78	0/0
	First upd. (win. opened): prim./graph. bl.	921/65	-/0
	First upd. (win. opened): delay	1005	1289
	Second update (win. closed): KB tr./KB graph. bl.	23/12	0/0
	Second upd. (win. closed): prim./graph. bl.	143/7	-/0
	Second upd. (win. closed): delay	156	0
	First interaction: overall delay	1161	1289
	Third update (win. opened): KB tr./KB graph. bl.	119/78	0/0
	Third upd. (win. opened): prim./graph. bl.	921/65	-/0
	Third upd. (win. opened): delay	1005	73
	Fourth update (list op.): KB tr./KB graph. bl.	44/23	0/0
	Fourth upd. (list op.): prim./graph. bl.	450/26	-/0
	Fourth upd. (list op.): delay	478	0
	Second interaction: overall delay	1483	73
	Fifth update (selection): KB tr./KB graph. bl.	15/8	0/0
	Fifth upd. (selection): prim./graph. bl.	347/24	-/0
	Fifth upd. (selection): delay	317	0
	Sixth update (apply preset): KB tr./KB graph. bl.	523/489	503/503
	Sixth upd. (apply preset): prim./graph. bl.	998/47	-/26
	Sixth upd. (apply preset): delay	1424	698
	Third interaction: overall delay	1741	698
	Overall delay (all the interactions)	4385	2060

9 Conclusion and Future Work

In this paper, a strategy aimed at transforming emerging Web OS architectures into complete desktop replacement solutions is presented. This strategy relies on an alternative virtualization paradigm, that exploits operating system-independent image processing techniques to extract a structured description of a generic application's graphics interface running on a remote desktop machine (possibly hosted by some hardware and/or software provider), which is later recreated within the Web OS environment.

By mixing remote computing solutions with Web-related technologies, the proposed approach is characterized by a high portability with respect to both client and server platforms, and allows to transparently reuse existing software in Web-based contexts without any local

installation or code re-writing. The designed architecture has the additional advantage that virtualized copies of applications can be tailored to user needs as well as to virtualization device characteristics, still preserving the same look and feel of applications natively available within the particular Web OS environment. Lastly, experimental results showed that, with respect to traditional virtualization solutions, the designed approach could give significant performance improvements in terms on both network bandwidth usage and interaction latency. In this way, the original application set of today's Web OSs can be effectively enriched, by providing users with a portable local-like interaction with applications they are accustomed to use on their traditional desktops.

Future work will be aimed at further investigating the interface analysis stage, by mixing data obtained by the promising image processing-based techniques considered in the current study with information that could be gathered from accessibility/automation-aware applications. This could allow to improve interface description accuracy and completeness, hence contributing at further extending the applicability of the designed approach. Moreover, the possibility to re-build customized interfaces at the client side could also affect and improve the usability of remote applications. For instance, new GUI structures could be created to enable the usage of complex applications on devices with limited visualization capabilities; in the same way, new interaction paradigms could replace "conventional" human-machine interaction ways. A well structured set of tests and a deep analysis of users feedback will be necessary to evaluate the impact of the proposed methodology on application usability.

Acknowledgements

This article is developed within the frame of the project LITES "Piattaforma Tecnologica Innovativa per lInternet of Things", which was co-funded by the Regione Piemonte, Italy.

References

1. J.P. Riganti and P.B. Schneck (1984), *Supercomputing*, IEEE Computer, vol. 17, no. 10, pp. 97–113.
2. M. Campbell-Kelly (2009), *The rise, fall, and resurrection of software as a service*, Communications of the ACM, vol. 55, no. 5, pp. 28–30.
3. J.P. Kanter (1998), *Understanding thin-client/server computing*, Microsoft Press, Redmond, WA, USA.
4. J. Grudin (2006), *The GUI shock: computer graphics and human-computer interaction*, ACM Interactions, vol. 13, no. 2, pp. 46–46.
5. G. Pallis (2010), *Cloud computing: the new frontier of Internet computing*, IEEE Internet Computing, vol. 14, no. 5, pp. 70-73.
6. K. Miller and M. Pegah (2007), *Virtualization: virtually at the desktop*, 35th ACM SIGUCCS Conference on User Services, pp. 255–260, ACM Press, USA.
7. S. Stegmaier, J. Diepstraten, M. Weiler and T. Ertl (2003), Widening the remote visualization bottleneck, 3rd International Symposium on Image and Signal Processing and Analysis, pp. 174–179, IEEE Computer Society Press, USA.
8. mental images RS_WP_310708 (2008), *RealityServer functional overview*, mental images GmbH, Berlin, Germany.
9. R.W. Scheifler and J. Gettys (1986), *The X Window system*, ACM Transactions on Graphics, vol. 5, no. 2, pp. 79–109.
10. T. Richardson, Q. Stafford-Fraser, K.R. Wood and A. Hopper (1998), *Virtual Network Computing*, IEEE Internet Computing, vol. 2, no. 1, pp. 33–38.

11. Microsoft MS.KB.186607 (2007), *Understanding the Remote Desktop Protocol (RDP)*, Microsoft Corporation, Redmond, WA, USA.
12. J. Golick (1999), *Network computing in the new thin-client age*, ACM netWorker, vol. 3, no. 1, pp. 30–40.
13. G. Lawton (2008), *Moving the OS to the Web*, IEEE Computer, vol. 41, no. 3, pp. 16–19.
14. A. Weiss (2005), *WebOS: say goodbye to desktop applications*, ACM netWorker, vol. 9, no. 4, pp. 18–26.
15. J. Walz and D.A. Grier (2009), *Time to push the cloud*, IT Professional, vol. 12, no. 5, pp. 14–16.
16. A. Allen (2009), *Palm webOS: Developing applications in JavaScript using the Palm MojoT framework*. O'Reilly Media, Inc., Sebastopol, CA, USA.
17. L. Shklar and R. Rosen (2003), *Web application architecture: Principles, protocols and practices*, John Wiley & Sons, Chichester, England.
18. S. Adee (2010), *Chrome the conqueror*, IEEE Spectrum, vol. 47, no. 1, pp. 34–39.
19. W. Yi and M.B. Blake (2010), *Service-oriented computing and cloud computing: Challenges and opportunities*, IEEE Internet Computing, vol. 14, no. 6, pp. 72–75.
20. F. Lamberti and A. Sanna (2008), *Extensible GUIs for remote application control on mobile devices*, IEEE Computer Graphics and Applications, vol. 28, no. 4, pp. 50–57.
21. TightVNC Documentation (2009), *TightVNC Documentation*, TightVNC Software.
22. WeirdX Documentation (2009), *WeirdX - Pure Java X Window system server under GPL*, JCraft.
23. Microsoft MS.WP.811522 (1998), *Microsoft Windows NT Server 4.0, Terminal Server edition: An architectural overview*, Microsoft Corporation, Redmond, WA, USA.
24. B.C. Cumberland, G. Carius and A. Muir (1999), *Microsoft Windows NT Server 4.0, Terminal Server edition: Technical reference*, Microsoft Press, Redmond, WA, USA.
25. Microsoft MS.WP.268349 (2008), *Windows Server 2008 Terminal Services*, Microsoft Corporation, Redmond, WA, USA.
26. A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham and C. Yoshikawa (1998), *WebOS: operating system services for wide area applications*, 7th Int'l Symposium on High Performance Distributed Computing, pp. 52–63, IEEE Computer Society Press, USA.
27. O. Krone and S. Schubiger (1999), *WebRes: Towards a Web operating system*, Kommunikation in Verteilten Systemen, pp. 418–429.
28. A. Vahdat, M. Dahlin, P. Eastham, C. Yoshikawa, T. Anderson and D. Culler (1997), *WebOS: Software support for scalable Web services*, 6th Workshop on Hot Topics in Operating Systems.
29. W. Bolosky, R.P. Draves, R.P. Fitzgerald, C.W. Fraser, M.B. Jones, T.B. Knoblock and R. Rashid (1997), *Operating system directions for the next millennium*, 6th Workshop on Hot Topics in Operating Systems, pp. 106–110.
30. D. Lewis (2006), *What is web 2.0?*, ACM Crossroads, vol. 13, no. 1, pp. 3–3.
31. R. Potter (1993), *Triggers: guiding automation with pixels to achieve data access*, Watch what I do: Programming by demonstration, MIT Press, Cambridge, MA, USA.
32. L.S. Zettlemoyer and R.S. Amant (1999), *A visual medium for programmatic control of interactive applications*, SIGCHI Conference on Human Factors in Computing Systems, pp. 199–206, ACM Press, USA.
33. R.S. Amant and L.S. Zettlemoyer (2000), *The user interface as an agent environment*, Int'l Conference on Autonomous Agents, pp. 483–490, ACM Press, USA.
34. D.S. Tan, D. Meyers and M. Czerwinski (2004), *WinCuts: manipulating arbitrary window regions for more effective use of screen space*, Int'l Conference on Human Factors in Computing Systems, pp. 1525-1528, ACM Press.
35. W. Stuerzlinger, O. Chapuis, D. Phillips and N. Roussel (2006), *User interface facades: towards fully adaptable user interfaces*, 19th ACM Symposium on User interface Software and Technology, pp. 309–318, ACM Press.
36. Microsoft UI Automation (2008), *Windows Presentation Foundation: UI Automation overview*, Microsoft Corporation, Redmond, WA, USA.
37. N. Souchon and J. Vanderdonckt (2003), *A review of XML-compliant User Interface Description*

Languages, 10th Int'l Workshop on Design, Specification, and Verification, pp. 377–391, Springer-Verlag, Berlin.
38. D. Thevenin, J. Coutaz and G. Calvary (2004), *A Reference Framework for the Development of Plastic User Interfaces. in Multiple User Interfaces*, pp. 29–49, John Wiley & Sons, England.
39. M.F. Ali, M.A. Perez-Quinones, and M. Abrams (2004), *Building Multi-Platform User Interfaces with UIML*, in A., S. and H., J. eds. Multiple User Interfaces, pp. 95–116, John Wiley & Sons, England.
40. A. Puerta and J. Eisenstein (2002), *XIML: A Common Representation for Interaction Data*, IUI2002: 6th International Conference on Intelligent User Interfaces, ACM, pp. 214–215.
41. Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. Lopez (2004), *UsiXML: a Language Supporting Multi-Path Development of User Interfaces*, 9th IFIP Working Conf. on Engineering for Human-Computer Interaction jointly with 11th Int. Workshop on Design, Specification, and Verification of Interactive Systems EHCIDSVIS' 2004 Hamburg, Germany, pp. 200–220.
42. G. Mori, F. Paternò, and C. Santoro (2004), *Design and Development of Multidevice User Interfaces through MultipleLogical Descriptions*, IEEE Transactions on Software Engineering, vol. 30, no. 8, pp. 507–520.
43. V. Bullard, K.T. Smith and M.C. Daconta (2001), *Essential XUL programming*, John Wiley & Sons, Chichester, England.
44. K. Gajos, D. Christianson, R. Hoffmann, T. Shaked, K. Henning, J.J. Long and D.S. Weld (2005), *Fast and robust interface generation for ubiquitous applications*, 7th International Conf. on Ubiquitous Computing, pp. 37–55, Springer, Berlin.
45. J. Nichols, B.A. Myers, M. Higgins, J. Hughes, T.K. Harris, R. Rosenfeld and M. Pignol (2002), *Generating remote control interfaces for complex appliances*, 15th ACM Symposium on User Interface Software and Technology, pp. 161–170, ACM Press, USA.
46. F. Lamberti and A. Sanna (2007), *A streaming-based solution for remote visualization of 3D graphics on mobile devices*, IEEE Transactions on Visualization and Computer Graphics, vol. 13, no. 2, pp. 247–260.
47. S. Stegmaier, M. Magallon and T. Ertl (2002), *A generic solution for hardware-accelerated remote visualization*, EG/IEEE TCVG Symposium on Visualization, pp. 87–94, Eurographics Association, Switzerland.
48. eyeOS Developer manual (2008), *eyeOS Developer manual*, eyeOS Project.
49. eyeOS User manual(2007), *eyeOS User manual*, eyeOS Project.
50. Microsoft UISpy (2008), *Windows Presentation Foundation Tools UI Spy (UISpy.exe)*, Microsoft Corporation, Redmond, WA, USA.