2010

Towards Providing Lightweight Access to Legacy Applications as Cloud-Based Services

Thimo Schulze
University of Mannheim, schulze@wifo.uni-mannheim.de

Christian Thum
University of Mannheim, thum@wifo.uni-mannheim.de

Markus Klems
Karlsruhe Institute of Technology, markus.klems@kit.edu

Towards Providing Lightweight Access to Legacy Applications as Cloud-Based Services

Thimo Schulze, Christian Thum

Chair in Information Systems III
University of Mannheim
Mannheim, Germany
Email: [schulze;thum]@wifo.uni-mannheim.de

Markus Klems

Institute of Applied Informatics and Formal Description Methods (AIFB)
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: markus.klems@kit.edu

Abstract

*Software as a Service solutions are available for certain business applications. However, many companies still rely on complex legacy applications for business-critical tasks. They cannot easily be accessed via the Internet and integrated into service-oriented landscapes. Re-programming or adaption of these legacy applications is both time-consuming and expensive. Remote access does not allow deep integration with other services and relies on proprietary software. We therefore propose the generic black-box approach REFLECTION (**Ref**urbishing **le**ga**c**y applica**tion**s) to dynamically rebuild the user interface of applications using native Web technologies. Users can thereby access these applications on-demand as cloud-based services. We also discuss usage scenarios, design and architecture considerations, as well as technical challenges.*

Keywords

Web-based Services, Cloud Computing, Software as a Service, User Interface, System Integration

INTRODUCTION

Over the last years, Software as a Service (SaaS) solutions running on cloud computing resources have been developed for various application areas. Different solutions are available online – the spectrum ranges from webmail, collaboration, and word processing to complex domains such as Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM). However, most companies still rely on legacy applications for their processes and services. These legacy applications have grown over decades and, since they constitute huge investments, they are still used even though better technologies are available (cf. Papazoglou and Ribbers 2006 p. 465). These very complex applications are often used for mission critical tasks in day-to-day business. Re-programming or adaption of these legacy applications would be both time-consuming and expensive. Furthermore, it is often not possible because of limited knowledge about the original program code, outdated or incomplete documentation and limited IT budgets.

Therefore, companies that use legacy applications cannot profit from many advantages that distributed application on cloud computing resources have. Cloud computing can be defined as virtualized, scalable computing resources that can be provisioned on demand as a service as well as software running on these resources (Armbrust et al. 2010; Vaquero et al. 2009). Legacy applications are often trapped in application silos and cannot easily be integrated into service-oriented architectures. Therefore, other approaches than traditional Enterprise Architecture Integration (EAI) are needed (cf. Alonso 2004 p. 123). A major problem is the lack of integration points between legacy applications and Software as a Service. Also, in a globalized world, a company would benefit from location-independent access to these applications and the ability to use collaboration functionality.

We therefore propose REFLECTION (**Ref**urbishing **le**ga**c**y applica**tion**s), a generic approach to dynamically rebuild the user interface (UI) of applications. With REFLECTION, the user logs into a website and starts a new application instance. He controls the application using the browser without needing access to or knowledge about the underlying architecture. The user does not have to install any plugins or extensions and still has a look-and-

feel similar to the legacy application. Relying only on functionality natively supported by modern Web browsers reduces the system requirements towards participating clients, which facilitates on-demand use and is characteristic for our lightweight approach (Thum and Schwind 2010). It is therefore applicable to various client devices like desktop computers, notebooks, tablets, netbooks, or smartphones. The underlying architecture runs on-demand on virtualized, scalable computer resources in the cloud.

This paper is organized as follows. In the next sections, related work on accessing legacy application over the Internet is described. Then, use cases and requirements for our approach REFLECTION are introduced and the underlying architecture is described. Finally, the paper concludes by discussing technical challenges and additional application areas.

RELATED WORK AND CURRENT TECHNOLOGY

The goal of this paper is to present a new method to make legacy applications accessible to heterogeneous end-user devices over the Internet. In principle, there are three existing generic approaches: First, the complete application code or parts of it can be transformed to Web technologies. Second, access to existing applications can be provided using gateways and wrappers with minimal changes in the original code. Third, users can use legacy systems via remote access or screen sharing software. In this section, we will give an overview of existing approaches. Additionally, the concept of "virtual appliances" is introduced as a basic mechanism for isolated, portable application packaging and hosting.

Code Transformation

The first possibility is to re-program entire applications or transform source code into new applications. One of the restrictions assumed for our approach REFLECTION is that access to the source code is limited and the existing application should not be changed. Therefore, the first approach is out of scope and only a basic overview is given.

Re-Programming. The advantage of re-programming a complete system is that technology and architecture can be changed and adapted to new business logics. Often, the new system is more flexible and easier to maintain. Since a completely new system can be expensive and time-consuming, it has to be evaluated if only some parts of the application need to be refined while others may remain unchanged.

Code-Conversion. For certain domains, tool-supported conversion of source code can significantly reduce complexity. These tools are able to analyze existing source code, convert it to a meta-language, and then transform it to modern languages like Java. For example, Douceur et al. (2008) introduce a browser plugin model that enables developers to adapt legacy code for use in rich Web applications. Their goal is to maintain security, performance, and OS-independence. Hofer and Fahringer (2007) present a toolkit that can convert existing software to deploy it on grid resources or on grid application services. It uses a semi-automatic transformation which includes manual refinement by developers.

Wrapping Approaches

In general, wrapping technologies can be divided into two categories. According to Weiderman et al. (1997) and Lucia et al. (2008), white-box wrapping requires access to source code and involves reverse engineering of individual application modules. Black-box transformation does not require access to source code and involves reverse engineering of interfaces or input/output data streams. Lucia et al. (2008) and Papazoglou and Ribbers (2006 p. 465ff) give a comprehensive overview over multiple existing approaches.

White-box Wrapping Approaches. One possibility is to use (Web)-service interfaces to access existing functionalities. Li and Qi (2004) present a Web-services-oriented wrapper generator that can be used to wrap legacy code as Web services. The goal is to re-use code in distributed problem-solving environments. Li et al. (2008) present a toolkit that can automatically wrap legacy software into services that can be published, discovered and reused in grid environments. They evaluate it by wrapping computer animation rendering code into a service that can be accessed by the Sun Grid Engine. Lucia et al. (2008) describe a comprehensive white-box wrapping approach that involves reengineering of the user interface using Web technology, the transformation of interactive legacy programs into batch programs, and the wrapping of the legacy programs.

Black-box wrapping approaches. This category contains approaches similar to REFLECTION. Black box wrapping does not change existing source code of a legacy application but tries to reproduce functionality with new user interfaces. Lin et al. (2004) and Hong et al. (2006) convert Windows-based applications into CORBA objects. Bovenzi et al. (2003) transform character-based user interfaces to any Web-based client device, for example a WAP mobile phone. Canfora et al. (2008) state that software systems modernization using Service Oriented Architectures (SOAs) and Web Services is a valuable option to extend the lifetime of mission-critical

legacy systems. They present a black-box modernization approach for exposing interactive functionalities of legacy systems as services.

Garlan et al. (2009) state that wrapping mechanisms can help, but often only work for a small part of problems or under narrow constraints. "For example, developers trying to integrate a legacy stand-alone application into a SOA often find that to 'wrap' the component in order to have a service interface, they must almost completely rewrite the application – for example, to decouple application code from its user interface" (Garlan et al. 2009).

Remote Access

Remote access can be defined as the ability to access computers or applications from a remote distance. It is often referred to as "Remote Desktop" or "Screen Sharing" and further distinguished into "Remote Controlling" or "Remote Administration". There are different remote desktop protocols, such as Virtual Network Computing (VNC) or the Windows Remote Desktop Protocol (RDP). Various free solutions based on VNC like RealVNC (www.realvnc.com), TightVNC (www.tightvnc.com), and UltraVNC (www.uvnc.com) are available and commercial remote access programs like GoToMyPC (www.gotomypc.com), LogMeIn (secure.logmein.com), Netviewer (www.netviewer.com), RemotePC (www.remotepc.com), or TeamViewer (www.teamviewer.com) offer rich functionality. Services like MokaFive (www.moka5.com) use remote access to centrally create, deliver, secure, and update fully-contained virtual environments and distribute them to thousands of users (cf. Mokafive 2010). Furthermore, there are approaches to combine remote access and Web services. For example, Zhang et al. (2008) propose a solution that uses VNC to give access to a legacy GUI and enable exchange of user operations. Interactions between users and legacy system can be integrated in a SOA.

The advantages of remote access include fast availability, easy usability, full control over applications, and unchanged look-and-feel. However, applications used via remote access cannot be integrated into existing processes and services. Since they still rely on physical or virtual computers and most of them are limited to specific operation systems, they cannot scale the screen resolution which makes them inappropriate for different devices like smartphones. Also, high bandwidth used for video-based transfer of screen contents can lead to delays.

Google recently announced "Chromoting", the possibility to run "legacy PC applications" in their browser based operation system Chrome OS. However, as of July 2010, "details on how Chromoting actually works are, unfortunately, scant" (Murphy 2010).

Virtual Appliances

An appliance is a device that delivers special-purpose services. Examples include digital video recorders, network routers, and so on. Different from a multi-purpose PC, an appliance comes with a complete hardware and software stack (firmware) focused on a small set of specialized services. Services provided by appliances are better isolated and usually better manageable than multiple services running on the same operating system. With a PC, the user is responsible for the proper functioning of services. With an appliance, the appliance manufacturer must test and ensure that the required services are working properly. If the appliance is connected to a network, its firmware can be updated and patched remotely. Operations are shifted from unprofessional users to professional operators.

The concept of a virtual appliance introduces an appliance that does not have a physical representation, but only a data representation (cf. Sapuntzakis et al. 2003; Sapuntzakis and Lam 2003). A virtual appliance can be set up as a virtual machine (VM) with special-purpose software packages. As such, a virtual appliance can be moved via network from one physical platform to another. Thus, virtual appliances are also more portable than physical appliances. They depend on the virtual machine monitor (hypervisor), though. Hypervisors can be installed on different hardware configurations and thereby decouple the virtual appliance from physical and operating system setups. A virtual appliance can interact through the usual I/O interfaces, such as computer keyboard, mouse, printer, and screen. The virtual appliance approach is related to wrapping. Access is usually given through remote login.

REFLECTION - BROWSER NATIVE GUI REENGINEERING

Motivation and Requirements

The generic approach REFLECTION introduced in this paper works for a large variety of enterprise applications. These include Customer Relationship Management (CRM) systems, Human Resource Management (HRM) tools, hospital information systems, Business Process Management (BPM) tools, Financial Accounting solutions, etc.

As a representative use case for this paper, we consider a Microsoft Windows application using standard control elements like menus, buttons, and text fields. For example, a goal could be to integrate existing cost estimation or

pricing tools into SaaS applications like Salesforce (www.salesforce.com) to achieve a better incorporation of this legacy software into processes or services of the company – or even as a perquisite for process automation.

SaaS running in the cloud promises many advantages, most prominently opening new markets for worldwide service delivery and enabling new pay-per-use and subscription-based licensing models. Customers can access software functions from different devices because applications are decoupled from technical infrastructure. It allows multiple users to access software at the same time (multi-tenancy) while ensuring higher service levels and fault tolerance (cf. Klems et al. 2009; Sun et al. 2010; Vouk 2008; Weinhardt et al. 2009).

Today, most SaaS applications offer general functionalities used by various clients. Some platforms for specialized business services have emerged. For example, Salesforce AppExchange is a marketplace for apps and services that extend Salesforce CRM and the force.com platform (sites.force.com/appexchange). However, easy integration of legacy applications remains impossible. As mentioned before, re-programming of legacy applications with employing technologies used by SaaS providers can be too expensive, time-consuming, complex, or difficult because of limited documentation.

For REFLECTION, we therefore address the following requirements:

- *No change of existing source code.* Since the source code of many legacy applications is no longer available or poorly documented, our approach should as a "black-box wrapping" work without any access to or change of existing source code.
- *Lightweight, ubiquitous access.* The applications should be accessible on various devices and be location-independent. No installation of browser-specific add-ons or plugins should be required. Computing or bandwidth requirements on the client device should be minimized.
- *Usability.* Users should have a look-and-feel similar to the legacy system without the need to learn using new user interfaces or processes.
- *Multi-tenancy.* Multiple users should be able to use instances of the application at the same time.
- *Integration.* Legacy applications should be integrated into modern Web applications, mashups, or SaaS products and thus, allow seamless exchange of information or data.

Existing approaches mentioned in the previous section are not capable of meeting these requirements. Re-programming, code-conversion, and white-box-wrapping all need access to source code. Most existing black-box approaches do not mention browser-based access. Allowing easy multi-tenancy with cloud computing services is not realized yet. Remote access allows world-wide access to applications while remaining usability and enabling multi-tenancy. However, remote access does not allow any integration into SaaS products because the user interface and the operating process remain unchanged. Most of them also need proprietary installations and have high bandwidth requirements for video-based transmission of desktop contents. Therefore, a different approach is needed which we introduce in the next section.

REFLECTION Workflow

In REFLECTION, clients initiate a service session by requesting a legacy application workspace. The request is routed to a compute cloud that triggers provisioning of a virtual machine (VM). The VM hosts the legacy application software. When the provisioning process has finished, the client is given access to the VM instance through a website representation of the legacy interface. Unlike remote access, the user does not directly access the VM operating system. Instead, an additional software component installed on the VM, the "Mediator", translates communication between website representation and the legacy application.

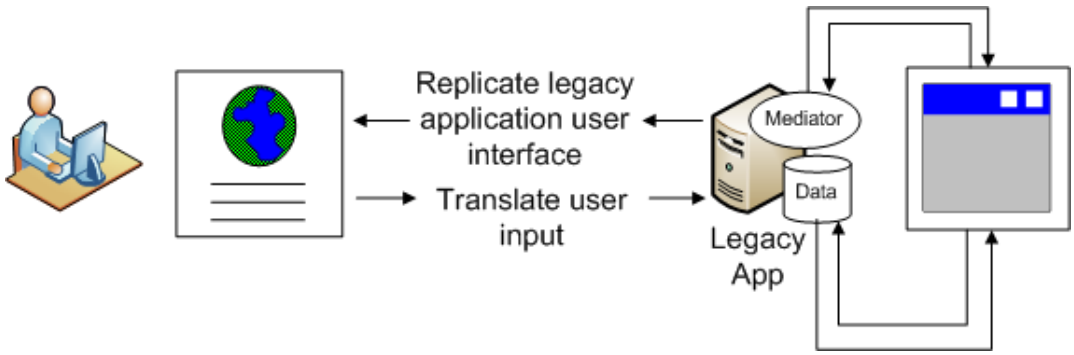

Figure 1: High-level Workflow Diagram

When the client clicks a button or performs other events on the website, these requests are routed to the VM instance and executed by the Mediator. The Mediator constantly observes the legacy application and reports application state changes, e.g. in response to a "button click" event. When the application state changes, e.g. a new window opens, the website changes as well. This high-level workflow is illustrated in Figure 1.

Software Architecture

We present a network-based application architecture based on the layered architectural style for multi-tiered client-server applications. The architecture is divided into presentation layer, communication layer, two application layers, and data layer. Moreover, the architecture is service-oriented, i.e. a lower layer provides services to an upper layer based on accepted standard interfaces and protocols. Our architecture proposes a thin client architecture where a client is a lightweight input/output device with the main capability of rendering graphical representations (e.g. a Web browser).

Figure 2 gives an overview of the software architecture. The responsibilities of architecture layers as well as the communication between layers are described in the following sections. We follow a black-box wrapping approach in combination with virtual appliances in a compute cloud. The overarching goal is to provide non-blocking near-real-time state synchronization between a lightweight user interface on the one side, and the state of a heavyweight legacy application on the other side.

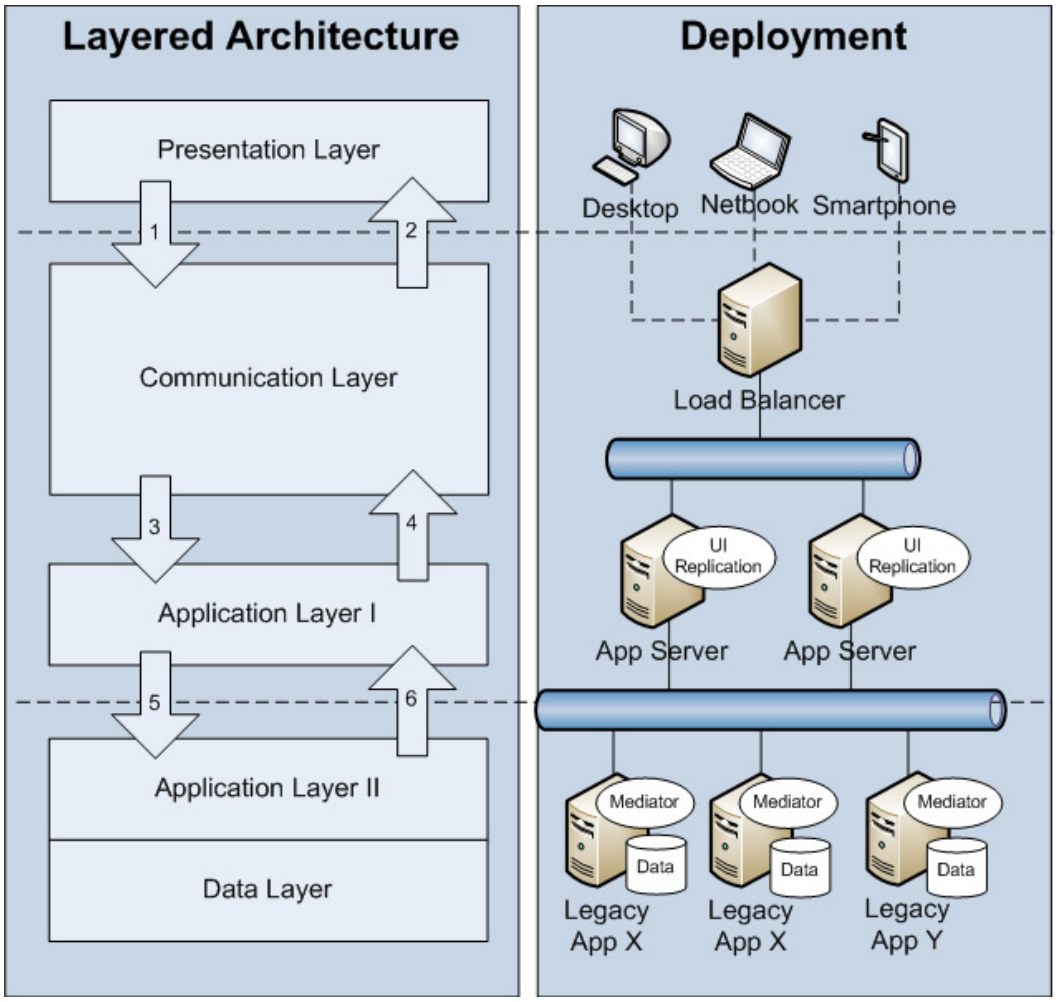

Figure 2: REFLECTION Software Architecture

Presentation Layer

The presentation layer shows a dynamically generated representation of the application state using the client's native user interface elements. For example, a Web browser client would render XML-based application state representations. A mobile device client with smaller screen, on the other hand, could represent the same application logic with a set of native user interface widgets for usability and performance reasons. However, since today's mobile device programming frameworks require source code compilation, dynamic creation of native widgets is currently not practicable. Therefore, our approach will focus on widely supported markup languages

and dynamic languages, like HTML and JavaScript, to create user interfaces. With this approach, legacy applications become accessible to a wide range of heterogeneous end-user devices. For the restructuring and visualization of applications on different screen sizes, existing research in the domain of model-driven UI development will be leveraged.

The representation of user interface elements is sent over the network in a serialized data interchange format, e.g. JavaScript Object Notation (JSON). When the serialized data arrives at the presentation layer, it serves as a basis for reconstructing the user interface using browser-native technologies. Today, JavaScript toolkits such as the Dojo Toolkit (www.dojotoolkit.org) or qooxdoo (qooxdoo.org) act as cross-browser wrappers, providing a solid basis for reconstructing the UI using browser-native technologies. Other server-centric approaches, such as the Google Web Toolkit (GWT), generate and send browser-specific representations based on previously identified client browser capabilities.

For scalability reasons, it can be beneficial to render the user interface elements entirely on the client side. This does not only take load off the server, which only acts as a forwarding proxy between browsers and the legacy application, but also enables the JavaScript toolkits to consider browser-specific differences that could not be accounted for when rendering the UI on a central server.

Communication Layer

A major challenge in the design of REFLECTION is the synchronization between the browser's user interface representation and the legacy application state. A detailed description of a mechanism enabling a consistent synchronization of client states is given by Thum et al. (2009). The communication layer interacts with the presentation layer using an asynchronous application message exchange protocol (cf. Rose 2001). Changes from the presentation layer are pushed to the application layer (1) and vice versa (2). The communication layer essentially implements an event-driven architecture with an incrementally scalable message bus. Access to the message bus is given via Web service interfaces.

Hypertext Transfer Protocol (HTTP) is the standard application protocol for Web-based communication. However, HTTP is optimized for stateless, synchronous request/response communication (cf. Fielding 2000). As a consequence, all communication has to be initiated by the client, which is not a favorable mechanism for real-time conversations (Dix 1997). Following a polling approach, the client continuously sends requests to the server. Upon receiving the server's response, the connection is closed and a new request is issued by the client. Unfortunately, polling has serious drawbacks. In the worst case, a server-side event occurs immediately after sending a response. In this case, latency equals the sum of the length of the polling interval plus the time that passes until the HTTP response of the last polling request has reached the client. Although interactivity has increased in Web 2.0 applications, there is still no simple way of sending events from server to client (cf. Ryan 2009).

Known solutions to overcome this limitation involve Web browser extensions, such as Flash XML Sockets, or Ajax software framework mechanisms, such as the event handler of Google Web Toolkit 1.6+. Recently, the term "Comet" has been coined, subsuming all techniques that allow a server to initiate client notification and send event notifications over HTTP to clients with negligible latency. These notifications can be sent in response to occurring events without explicit polling. In future, the currently emerging HTML5 Web Sockets standard might provide an implementation alternative. The Web Sockets API and protocol are a particularly attractive as they promise a standardized, application-independent mechanism to establish a bi-directional communication channel between Web clients and servers (cf. Hickson 2010).

Application Layer I: User Interface Generator

The application layer is comprised of two functionally separate areas: user interface generation and legacy application hosting. The upper application layer acts as a gateway that receives user requests from the communication layer (3) via a Web service interface. It dynamically generates user interfaces and serves them to the communication layer (4). We use the Front Controller pattern (Fowler 2003) to channel user commands from the upper application layer to the lower application layer (5) where a handler selects the most appropriate command implementation (Figure 3 (b)). Conversely, user interface generation is triggered by the lower application layer and pushed up to the upper application layer (6) using the Event Collaboration pattern (Figure 3 (a)).

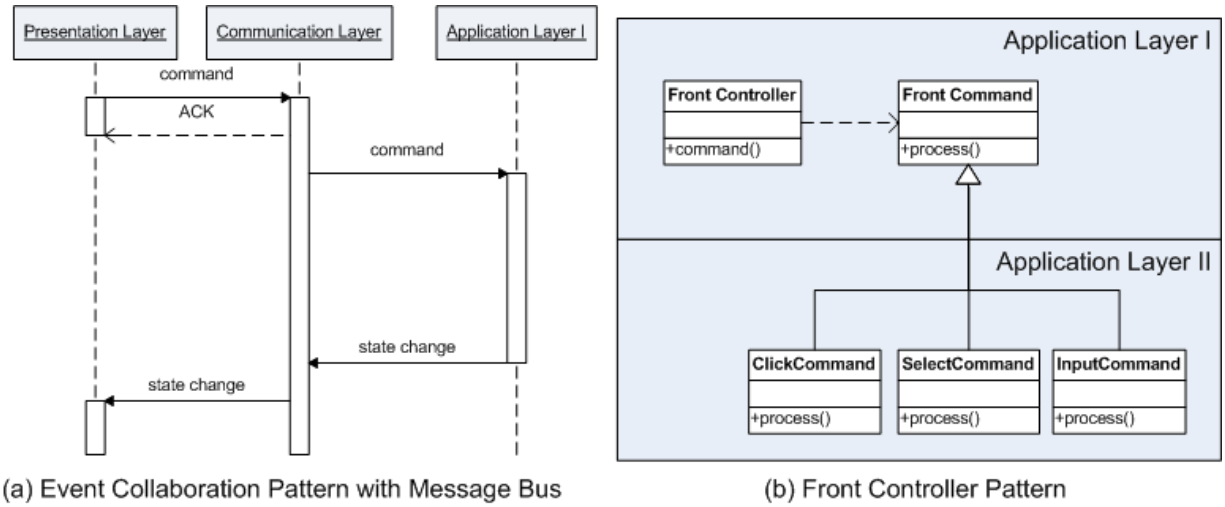

Figure 3: Front Controller pattern and Event Collaboration pattern in the context of the REFLECTION architecture

Application Layer II: Legacy Application and Mediator

The lower application layer is responsible for legacy application hosting and user interaction processing. Legacy applications are deployed on virtual machines in a compute cloud. As explained above, bundles of VM-based operating systems with special-purpose software are known as "virtual appliances" (Sapuntzakis et al. 2003). Our approach uses virtual appliances that bundle pre-installed and pre-configured legacy application software with an appropriate operating system environment on a sufficiently powerful single-node VM. Virtual appliances come with a pre-installed Mediator software component which is described in more detail in the next section.

Amazon's Elastic Compute Cloud (EC2) currently offers VM instances on a vertical scale starting with small instances (1.7 GB of memory, 1 virtual core, 160 GB of local instance storage, and 32-bit platform) up to high-CPU instances, high-memory instances and cluster compute instances (23 GB of memory, 33.5 virtual cores, 1690 GB of local instance storage, 64-bit platform, and 10 Gigabit Ethernet connection). EC2 supports a growing list of operating systems, including Red Hat Enterprise Linux, Fedora Linux, Gentoo Linux, Debian Linux, Ubuntu Linux, openSUSE Linux, OpenSolaris, Oracle Enterprise Linux, and Windows Server 2003/2008.

Multi-tenancy is realized by hosting a dedicated VM for each legacy application user, thus isolating each legacy application workspace. Authentication and session handling, including request/response routing, is under control of the communication layer. When an end-user triggers the request to use a legacy application workspace, a dedicated VM image is instantiated on-demand by using the compute cloud's capabilities to automate resource provisioning. It has to be noted that this approach currently only works for single tier legacy applications. If multiple users need access to the same application or underlying database, reengineering of the entire application could be necessary and our black box approach might not feasible.

Data Layer

The data layer is tightly coupled with the lower application layer. A legacy application might for example persist the session state to the local file system or to a remote relational database server connected via language-dependent connectors, such as Java Database Connectivity (JDBC) drivers. Although tight coupling is usually not a desirable design decision, as it limits horizontal scalability, we believe that achieving loose coupling would require substantial manual re-engineering of application code – which precisely what we wanted to avoid with our approach.

MEDIATOR

The Mediator is one of the central architectural components responsible for inspecting the application under consideration aiming to extract layout information that enables the reconstruction of the graphical user interface. Such information comprises the position and size of any window, command buttons or other controls as well as specific values of these controls. Figure 4 illustrates the REFLECTION approach.

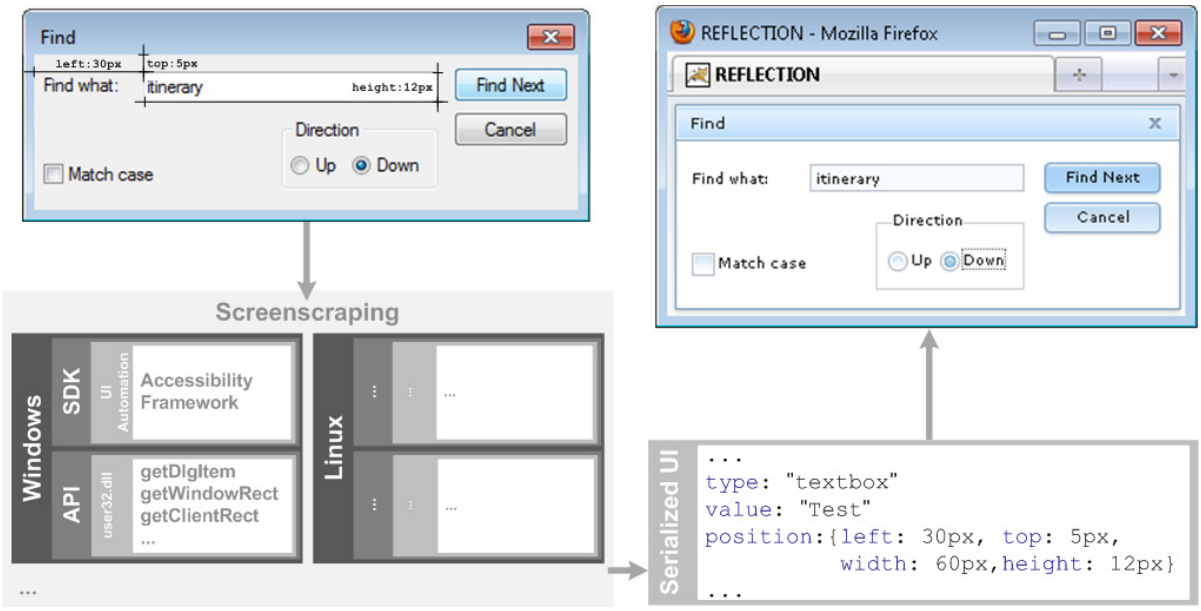

Figure 4: Extracting layout information from Windows applications and rebuilding in browser

The top section of the figure shows a standard Microsoft Windows dialog with basic control elements. By using native system libraries, information concerning the graphical user interface of any running process can be retrieved. This step is referred to as screen scraping and is depicted in the lower section of Figure 4. In our Microsoft Windows test environment, the system library (user32.dll) contains all methods relevant for GUI inspection. An application that demonstrates the feasibility of this approach is "The Customiser" (www.wanga.com/cu.php). Alternatively, the accessibility framework contained in the Microsoft Windows UI Automation library, that is part of the Microsoft Windows SDK, can be used. It allows developers to view the structure of an application's user interface, its property values, and raised events. For different system environments such as Linux or Mac OS, separate Mediators realizing screen scraping functionality on the basis of native APIs have to be implemented. As the scraped layout and UI information of the legacy application is then serialized to a standard data interchange format, other components of REFLECTION remain unchanged. As the serialized UI information has to be processed within Web browsers, either XML or JSON are applicable. Since parsing by the JavaScript Engine is faster, relying on JSON as illustrated in the bottom right part of the figure appears more adequate than using XML.

As the UI of the legacy application may change in response to user actions (mouse and keyboard input) implementing a monitoring mechanism is necessary to ensure a synchronized view on all clients. We identify two strategies which can be used in this context: periodic and interceptive.

Following a *periodic strategy,* the legacy application is monitored for UI changes within certain time intervals. The difficulty is to determine the optimal length of these intervals balancing between responsiveness, UI synchronization, and resource consumption. Whereas a long interval might result in slower synchronization or the skipping of UI changes, a short interval consumes more resources and hence impacts the overall system performance. The polling interval could be dynamically adapted in response to the replication of user actions that are likely to trigger UI changes.

The *interceptive strategy* takes advantage of a deeper understanding of the legacy environment's UI system. Intercepting the legacy application's calls to native system libraries by setting hooks on UI-specific APIs (e.g. createWindow, setWindowText) enables synchronization of UI changes with negligible latency.

CONCLUSION AND FUTURE WORK

Many companies still use legacy applications for business-critical processes. Making these applications available over the Internet and integrating them into existing service-oriented architectures or SaaS solutions is an important challenge. We introduced REFLECTION (Refurbishing legacy applications), a novel black-box approach that extracts the information from legacy applications and replicates the user interface for heterogeneous devices using technology of modern Web browsers. Since no knowledge of the legacy source code is required and the layout information is extracted and processed in a standardized way, this generic approach works in various areas. Users only need a modern Web browser with no additional plugins for world-wide access to the application while retaining a similar look-and-feel. With minimal effort, multiple users can access instances of the same application

simultaneously. Most important, since the application is now operated in the Web browser, it can be integrated with other Web applications in mashups or used seamlessly to exchange data or information with SaaS services.

The implementation of the approach poses some technical challenges. Since HTTP is limited to synchronous request/response communication, new technologies like Comet or HTML 5 are needed for near-real-time synchronization between legacy application server and client. It has to be assured that the view in the Web browser and the state of the legacy application converge to consistency within a short time frame. The biggest challenge is to make the approach so general that the dynamic UI generation can be used for most applications with minimal manual adjustment and no custom widgets. Procedures for handling legacy elements that are not available in some browsers (like high resolution diagrams or charts) need to be implemented. Therefore, focusing on basic form-based applications first seems reasonable.

Also, many cloud providers only allow a fixed set of operating systems. Especially Windows desktop operation systems like Windows XP or Windows 7 are currently not available due to licensing issues. As part of ongoing work we are therefore researching which legacy applications support the installation or migration on the available virtualized resources. The question for what kind of applications the REFLECTION approach is most effective, will also be addressed in future work.

In future work, we will also test the approach in more complex case studies. For example, a CRUD (Create, Read, Update and Delete) example could be used in order to analyze the effect of transferring and accessing data. In particular the latency has to be considered in such cases.

Besides the task of making legacy applications available over the Internet and integrating them into existing services, the architecture can also be used for another domain. Software vendors can use REFLECTION for providing existing software products as Software as a Service to clients. They need an additional business model layer and have to define pricing models. Clients can then get access to instances of the legacy software over their Web browser paying only based on usage or a subscription model. Since the additional effort needed for providing software over the Internet using this generic approach is low, it is especially relevant for specialized applications where new developments would be unprofitable.

REFERENCES

Alonso, G. 2004. *Web services: concepts, architectures and applications*, Springer.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. 2010. "A view of cloud computing," *Commun. ACM* (53:4), pp. 50-58.

Bovenzi, D., Canfora, G., and Fasolino, A. R. 2003. "Enabling Legacy System Accessibility by Web Heterogeneous Clients," In *Seventh European Conference on Software Maintenance and Reengineering*Los Alamitos, CA, USA: IEEE Computer Society, pp. 73.

Canfora, G., Fasolino, A. R., Frattolillo, G., and Tramontana, P. 2008. "A wrapping approach for migrating legacy system interactive functionalities to Service Oriented Architectures," *Journal of Systems and Software* (81:4), pp. 463-480.

Dix, A. 1997. "Challenges for Cooperative Work on the Web: An Analytical Approach," *Comput. Supported Coop. Work* (6:2-3), pp. 135-156.

Douceur, J. R., Elson, J., Howell, J., and Lorch, J. R. 2008. "Leveraging Legacy Code to Deploy Desktop Applications on the Web," In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 339-354.

Fielding, R. 2000. "Architectural styles and the design of network-based software architectures," Dissertation.

Fowler, M. 2003. *Patterns of enterprise application architecture*, Addison-Wesley.

Garlan, D., Allen, R., and Ockerbloom, J. 2009. "Architectural Mismatch: Why Reuse Is Still So Hard," *IEEE Softw.* (26:4), pp. 66-69.

Hickson, I. 2010. "The WebSocket API Editor's Draft 15 June 2010," Retrieved July 18th from *http://dev.w3.org/html5/websockets/*.

Hofer, J., and Fahringer, T. 2007. "The Otho Toolkit – Synthesizing tailor-made scientific grid application wrapper services," *Multiagent and Grid Systems* (3:3), pp. 281-298.

Hong, Z., Lin, J., Jiau, H. C., Fang, G., and Chiou, C. W. 2006. "Encapsulating windows-based software applications into reusable components with design patterns," *Information and Software Technology* (48:7), pp. 619-629.

Klems, M., Nimis, J., and Tai, S. 2009. "Do Clouds Compute? A Framework for Estimating the Value of Cloud Computing," In *Designing E-Business Systems. Markets, Services, and Networks*, pp. 110-123.

Li, M., and Qi, M. 2004. "Leveraging legacy codes to distributed problem-solving environments: a Web services approach," *Software: Practice and Experience* (34:13), pp. 1297-1309.

Li, M., Yu, B., Qi, M., and Antonopoulos, N. 2008. "Automatically wrapping legacy software into services: A grid case study," *Peer-to-Peer Networking and Applications* (1:2), pp. 139-147.

Lin, J., Hong, Z., Fang, G., Jiau, H. C., and Chu, W. C. 2004. "Reengineering windows software applications into reusable CORBA objects," *Information and Software Technology* (46:6), pp. 403-413.

Lucia, A. D., Francese, R., Scanniello, G., and Tortora, G. 2008. "Developing legacy system migration methods and tools for technology transfer," *Softw. Pract. Exper.* (38:13), pp. 1333-1364.

Mokafive. 2010. "Products Overview | MokaFive," Retrieved July 18th from *http://www.moka5.com/products/products-overview.php*.

Murphy, D. 2010. "Chrome OS to Support "Legacy" PC Apps Via "Chromoting" | News & Opinion | PCMag.com," Retrieved July 16th from *http://www.pcmag.com/article2/0,2817,2364969,00.asp*.

Papazoglou, M., and Ribbers, P. 2006. *E-business: organizational and technical foundations*, John Wiley.

Rose, M. 2001. "RFC3117 - On the Design of Application Protocols," Retrieved July 18th from *http://www.faqs.org/rfcs/rfc3117.html*.

Ryan, R. 2009. "Google I/O 2009 - Best Practices for Architecting GWT App," Retrieved July 18th from *http://www.youtube.com/watch?v=PDuhR18-EdM*.

Sapuntzakis, C., Brumley, D., Chandra, R., Zeldovich, N., Chow, J., Lam, M. S., and Rosenblum, M. 2003. "Virtual Appliances for Deploying and Maintaining Software," In *Proceedings of the 17th USENIX conference on System administration*, San Diego, CA: USENIX Association, pp. 181-194.

Sapuntzakis, C., and Lam, M. S. 2003. "Virtual appliances in the collective: a road to hassle-free computing," In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, Lihue, Hawaii: USENIX Association, pp. 10-10.

Sun, W., Zhang, K., Chen, S., Zhang, X., and Liang, H. 2010. "Software as a Service: An Integration Perspective," In *Service-Oriented Computing – ICSOC 2007*, pp. 558-569.

Thum, C., and Schwind, M. 2010. "synchronite—A Service for Real-Time Lightweight Collaboration," Presented at the 5th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'10), Fukuoka, Japan.

Thum, C., Schwind, M., and Schader, M. 2009. "SLIM—A Lightweight Environment for Synchronous Collaborative Modeling," In *Model Driven Engineering Languages and Systems*, pp. 137-151.

Vaquero, L. M., Rodero-Merino, L., Caceres, J., and Lindner, M. 2009. "A break in the clouds: towards a cloud definition," *SIGCOMM Comput. Commun. Rev.* (39:1), pp. 50-55.

Vouk, M. 2008. "Cloud computing — Issues, research and implementations," Presented at the 30th International Conference on Information Technology Interfaces, 2008. ITI 2008, pp. 31-40.

Weiderman, N., Northrop, L., Smith, D., Tilley, S., and Wallnau, K. 1997, June. "Implications of Distributed Object Technology for Reengineering," Technical Report.

Weinhardt, C., Anandasivam, A., Blau, B., Borissov, N., Meinl, T., Michalk, W., and Stößer, J. 2009. "Cloud-Computing," *WIRTSCHAFTSINFORMATIK* (51:5), pp. 453-462.

Zhang, B., Bao, L., Zhou, R., Hu, S., and Chen, P. 2008. "A Black-Box Strategy to Migrate GUI-Based Legacy Systems to Web Services," Presented at the 2008 IEEE International Symposium on Service-Oriented System Engineering (SOSE), Jhongli, Taiwan, pp. 25-31.

COPYRIGHT