Web Browser as an Application Platform

Antero Taivalsaari and Tommi Mikkonen
Sun Microsystems Laboratories
P.O. Box 553 (TUT)
FIN-33101 Tampere, Finland
firstname.lastname@sun.com

Dan Ingalls and Krzysztof Palacz
Sun Microsystems Laboratories
16 Network Circle, MPK 16
Menlo Park, CA 94025, U.S.A.
firstname.lastname@sun.com

Abstract

For better or worse, the web browser has become a widely used target platform for software applications. Desktop-style applications such as word processors, spreadsheets, calendars, games and instant messaging systems that were earlier written for specific operating systems, CPU architectures or devices are now written for the World Wide Web, to be used from a web browser. In this paper we summarize our experiences in using the web browser as a target platform for real applications. As a concrete example, we use the Sun™ Labs Lively Kernel, a system that implements an exceptionally interactive web programming environment running in a web browser without any plug-in components. Based on this work, we analyze the limitations, challenges and opportunities related to the web browser as an application platform.

1. Introduction

The widespread adoption of the World Wide Web has fundamentally changed the landscape of software development. In the past few years, the Web has become a popular deployment environment for new software systems and applications. We believe that in the near future the vast majority of new software applications will be written for the Web, instead of conventional target platforms such as specific operating systems, CPU architectures or devices.

In general, the software industry is currently experiencing a paradigm shift towards web-based software. In the new era of web-based software, applications live on the Web as services. They consist of data, code and other resources that can be located anywhere in the world. Furthermore, they require no installation or manual upgrades. Ideally, applications should also support user collaboration, i.e., allow multiple users to interact and share the same applications and data over the Internet.

In the era of web-based software, the web browser will take an ever more encompassing, central role in our lives. Among other things, the web browser will take over many roles that conventional operating systems used to have in serving as a launchpad and a host platform for applications when they are run. In the eyes of the average computer user, the web browser will effectively be the *de facto* operating system.

In this paper[1] we summarize our experiences in using a regular web browser as a platform for real, desktop-style applications. As a concrete example, we use the *Sun™ Labs Lively Kernel* (see http://research.sun.com/projects/lively/) – a system that pushes the limits of the web browser by implementing an exceptionally interactive web programming environment that runs in a web browser without installation or any plug-in components whatsoever. The absence of browser plug-ins makes the Lively Kernel different from other web application development systems such as *Adobe AIR* (http://www.adobe.com/products/air/) or *Microsoft Silverlight* (http://www.microsoft.com/silverlight/). Based on this work, we analyze the limitations, challenges and opportunities related to the web browser and web applications more generally. We also provide a number of recommendations for future improvements.

The structure of this paper is as follows. In Section 2, we provide a historical summary of the evolution of the Web, focusing especially on the ongoing transition from web pages towards web applications. We also provide an overview of the Sun Labs Lively Kernel – a flexible web programming environment designed at Sun Labs. In Section 3, we summarize our experiences in using the web browser as an application platform, taking a look at the various issues that we have discovered. In Section 4, we provide suggestions for future improvement. Section 5 concludes the paper.

[1] An earlier version of this paper has been published as Sun Labs Technical Report TR-2008-175, January 2008.

2. From Web Pages to Web Applications

Compared to how dramatically web usage has increased since the 1990s, it is remarkable how little the web browser has changed since it was introduced. For instance, the common navigation features, such as the "*back*", "*forward*" and "*reload*" buttons of the browser, were present already in the early versions of Mosaic and Netscape Navigator. In contrast, the way the Web is used has evolved constantly from the early days. In the following, we provide a brief summary of the evolution of web usage.

2.1. Evolution of Web Usage

The World Wide Web has undergone a number of evolutionary phases. Initially, web pages were simple textual documents with limited user interaction capabilities based on hyperlinks. Soon, graphics support and form-based data entry were added. Gradually, with the introduction of DHTML [1] – the combination of HTML, Cascading Style Sheets (CSS), the JavaScript scripting language [2], and the Document Object Model (DOM) – it became possible to create interactive web pages with built-in support for advanced graphics and animation. Numerous plug-in components – such as Flash, RealPlayer and Shockwave – were then introduced to make it possible to build web pages with visually rich, interactive multimedia content. At the high level, the evolution of web pages has advanced from simple, "classic" web pages with text and static images only to animated multimedia pages with plug-ins to *Rich Internet Applications* (RIA). Below we provide a summary of the three main phases in the evolution of the Web.

In the first phase, web pages were truly *pages*, i.e., page-structured documents that contained primarily text with some interspersed static images, without animation or any interactive content. Navigation between pages was based simply on hyperlinks, and a new web page was loaded from the web server each time the user clicked on a link. There was no need for asynchronous network communication or any advanced protocols between the browser and the web server. Some pages were presented as *forms*, with simple textual fields and the possibility to use basic widgets such as buttons, radio buttons or pull-down menus.

In the second phase, web pages became increasingly interactive, with animated graphics and plug-in components that allowed richer content to be displayed. This phase coincided with the commercial takeoff of the Web, when companies realized that they could create commercially valuable web sites by displaying advertisements or by selling merchandise or services over the Web. Navigation was no longer based solely on links, and communication between the browser and the server became increasingly advanced. The JavaScript scripting language, introduced in Netscape Navigator version 2.0B in December 1995, made it possible to build animated, interactive content more easily. The use of plug-in components such as Flash, Quicktime, RealPlayer and Shockwave spread rapidly, allowing advanced animations, movie clips and audio tracks to be inserted in web pages. In this phase, the Web started moving in directions that were unforeseen by its designers, with web sites behaving more like multimedia presentations rather than conventional pages. Content mashups[2] and web site cross-linking became increasingly popular.

Today, we are in the middle of another major evolutionary step towards desktop-style web applications, also known as *Rich Internet Applications* or simply as *web applications*. The technologies intended for the creation of such applications are also often referred to collectively as "*Web 2.0*" technologies. Fundamentally, Web 2.0 technologies combine two important characteristics or features: *collaboration* and *interaction*. By *collaboration*, we refer to the "social" aspects that allow a vast number of people to collaborate and share the same data, applications and services over the Web. However, an equally important, but publicly less noted aspect of Web 2.0 technologies is *interaction*. Web 2.0 technologies make it possible to build web sites that behave much like desktop applications, for example, by allowing web pages to be updated one user interface element at a time, rather than requiring the entire page to be updated each time something changes. Web 2.0 systems often eschew link-based navigation and utilize direct manipulation techniques familiar from desktop-style applications instead. Furthermore, some systems offer application development capabilities as built-in features. For instance, the Facebook web site (http://www.facebook.com/) has its own application description language that can be used for creating web applications for Facebook pages.

The three phases discussed above are not mutually exclusive. Rather, web pages representing all three phases coexist on the Web today. The majority of commercial web pages today represent the second phase. However, the trend towards web applications is becoming increasingly common, with new web

[2] In web terminology, a *mashup* is a web site that combines content from more than one source (from multiple web sites) into an integrated experience.

application development technologies and systems being introduced frequently.

2.2. General Observations and Trends

In analyzing the web application development technologies mentioned above, it quickly becomes obvious that all these technologies are still rather different from each other. However, there are some common themes that have started to emerge.

Trend toward dynamic languages [3]. Most of the systems above rely on dynamic, interpreted languages at least at some level. In some systems, such as Ajax (*http://www.ajaxian.com/*) or Ruby on Rails (RoR) (*http://www.rubyonrails.org/*), applications are written entirely in a dynamic language – JavaScript and Ruby, respectively. Other systems, such as Google Web Toolkit (GWT) (*http://code.google.com/webtoolkit/*), rely on a dynamic language (JavaScript) for program execution inside the web browser.

Technology mashups. Many current web systems are hybrid solutions in the sense that they combine various existing, sometimes previously unrelated technologies. For instance, Ajax [4] is a combination of a number of existing technologies – HTML, CSS, DOM, JavaScript, asynchronous HTTP networking and XML protocols – rather than a uniform, coherent application platform. In this regard, these systems resemble the content mashups that are common on the Web today.

Dependence on tools. Most of the web systems are heavily dependent on tools and integrated development environments. For instance, Ruby on Rails introduces a set of naming conventions that are automatically applied by the development tools.

Another general observation about web application development today is that they often violate well-known software engineering principles. For instance, the JavaScript language has very limited support for modularity or information hiding. We have summarized our observations in this area in more detail in another paper [5].

2.3. Sun Labs Lively Kernel

At Sun Labs, we have developed a new web programming environment called the *Sun Labs Lively Kernel*. The Lively Kernel supports desktop-style applications with rich graphics and direct manipulation capabilities, but without the installation or upgrade hassles that conventional desktop applications have. The system and the applications written for it run in a regular web browser without installation or plug-in components. The system even includes development tools that can be used inside the system itself.

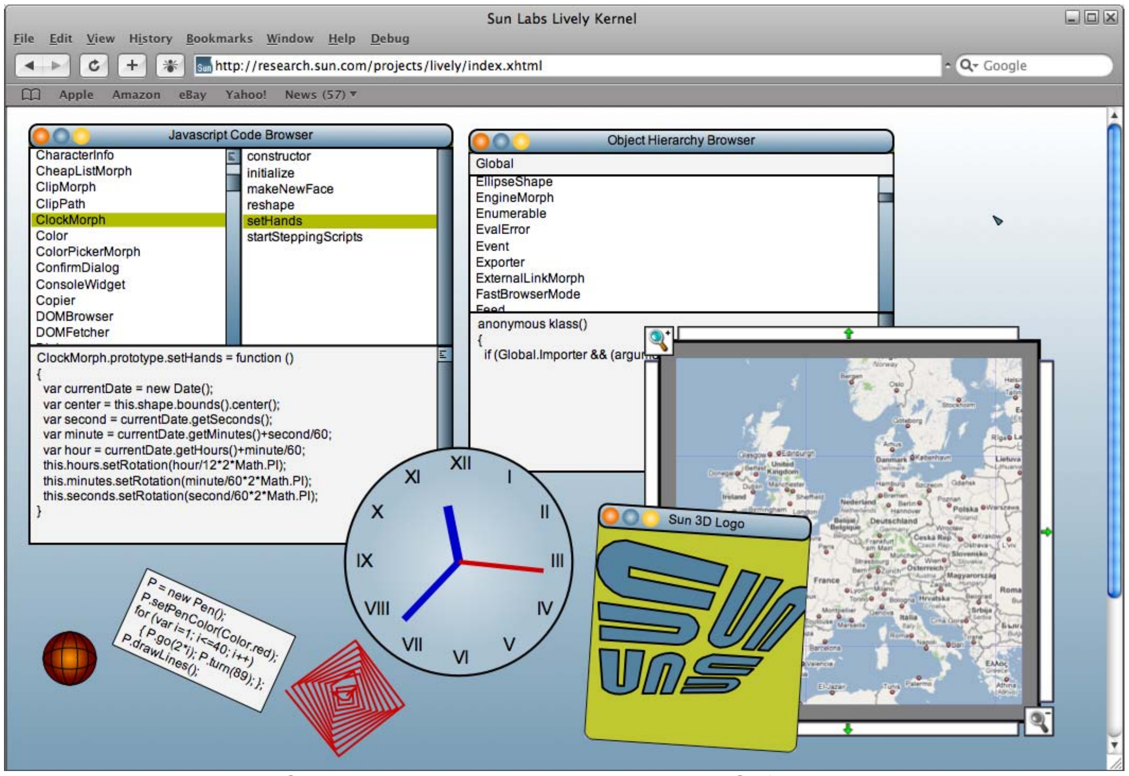
The Lively Kernel is built around the following three assumptions:

1. The World Wide Web is the new target platform.
2. The Web Browser is the new operating system.
3. JavaScript is the *de facto* programming language of the Web.

For the purposes of this paper, the Lively Kernel is especially interesting because the system pushes the limits of the web browser as an application platform further than any other system. In addition to supporting desktop-style applications that can run in a web browser, the Lively Kernel can also function as an integrated development environment (IDE), making the whole system self-supporting and able to improve and extend itself dynamically. Yet the entire system requires nothing more for its execution than a web browser. A screenshot of the system has been provided in Figure 1.

A key difference between the Lively Kernel and other systems in the same area is our focus on *uniformity*. Our goal is to build a platform using a minimum number of underlying technologies. This is in contrast with many current web technologies that utilize a diverse array of technologies such as HTML, CSS, DOM, JavaScript, PHP, XML, and so on. In the Lively Kernel we attempt to do as much as possible using a single technology: JavaScript. We have chosen JavaScript primarily because of its ubiquitous availability in the web browsers today and because of its syntactic similarity to other highly popular languages such as C, C++ and Java. However, we also want to leverage the dynamic aspects of JavaScript, especially the ability to modify applications at runtime. Such capabilities are an essential ingredient in building a malleable web programming environment that allows applications to be developed interactively and collaboratively. In some ways, the system illustrates that the entire computer desktop, including all the commonly used tools and applications, can be moved to the Web and run in a web browser as well.

From the technical viewpoint, the Lively Kernel system consists of the following four components.



```
ClockMorph.prototype.setHands = function ()
{
 var currentDate = new Date();
 var center = this.shape.bounds().center();
 var second = currentDate.getSeconds();
 var minute = currentDate.getMinutes()+second/60;
 var hour = currentDate.getHours()+minute/60;
 this.hours.setRotation(hour/12*2*Math.PI);
 this.minutes.setRotation(minute/60*2*Math.PI);
 this.seconds.setRotation(second/60*2*Math.PI);
}
```

```
anonymous klass()
{
  if (Global.Importer && (argu
```

```
P = new Pen();
P.setPenColor(Color.red);
for (var i=1; i<=40; i++)
  { P.go(2*i); P.turn(89); };
P.drawLines();
```

Figure 1. Sun Labs Lively Kernel running in the Safari web browser

1. The JavaScript programming language. We have used the JavaScript programming language, a facility available in all commercial web browsers today, as a fundamental building block for the rest of the system.

2. Asynchronous HTTP networking. All the networking operations in the Lively Kernel are performed asynchronously using the *XMLHttpRequest* feature familiar from Ajax. The use of asynchronous networking is critical so that all the networking requests can be performed in the background without impairing the interactive response of the system.

3. Morphic user interface framework and widgets. The Lively Kernel is built around a rich user interface framework called *Morphic* [6]. The Morphic framework consists of about 10,000 lines of uncompressed JavaScript code that is downloaded to the web browser when the Lively Kernel starts.

4. Built-in tools for developing, modifying and deploying applications on the fly. The Morphic UI framework includes tools – such as a class browser and object inspector – that can be used for developing, modifying and deploying applications from within the Lively Kernel system itself. These features have been implemented using the reflective capabilities of the JavaScript programming language, and can therefore be used inside the web browser without any external tools or IDEs. Furthermore, it is possible to export objects or entire web pages, so that applications written inside the Lively Kernel can also be run as standalone web pages.

Given these additional capabilities, and the fact that the system can run in an ordinary web browser, the Lively Kernel is an interesting research vehicle for studying the capabilities of the web browser as an application platform. The Lively Kernel also serves as a testbed for studying the use of the JavaScript language as a general-purpose programming language [7]. Some of the work that we have done with JavaScript could even be described as systems programming. In general, the Lively Kernel proves that there is a lot of latent, unleashed power in the web browser; for instance, the true potential of JavaScript as a dynamic, reflective programming language inside the browser has not been fully utilized so far.

3. Experiences and Observations

In this section we provide a summary of our experiences with the web browser as a host platform for applications. We have categorized our experiences and observations into usability and user interaction issues, networking and security issues, interoperability and compatibility issues, development and testing issues, deployment issues, and performance issues. Many of these issues have already been summarized in an earlier paper, although from a slightly different perspective [5].

3.1. General Observations and Trends

The browser I/O model is poorly suited to desktop-style applications. In current web technologies a scripting language and other external components communicate with the web browser primarily via the Document Object Model (DOM). DOM is effectively a large tree data structure that allows external tools to access the data that is displayed in the browser, typically by reading and modifying the attributes that represent the graphical objects on the screen. In addition to tweaking the DOM tree, an external tool such as a scripting language can construct HTML pages as strings (containing HTML markup) on the fly and then send those strings to the browser with the expectation that the browser will update its screen accordingly. This is cumbersome compared to traditional systems in which programs can manipulate the screen directly by using a graphics API that supports direct drawing and direct manipulation.

The page-based display update model of the web browser is also an impediment to application usability, as it is dramatically less interactive than the direct manipulation user interfaces that were in widespread use in desktop computers already in the 1980s. With Ajax and other Web 2.0 technologies supporting asynchronous network communication, the page-based update model is gradually being replaced with a finer-grained interaction model, but it is still hard to implement user interaction capabilities that would be on a par with desktop applications.

The semantics of many browser features are unsuitable for applications. The web browser has a number of historical features that have poorly defined semantics for applications. Consider the '*reload*', '*stop*', '*back*' and '*forward*' buttons, for instance. While such navigational features make sense when viewing documents and forms, these features have unclear semantics for applications that have a complex internal state and highly dynamic interaction with the web server. For example, it is difficult to define meaningful semantics for an online stock trading or a banking application's response to the '*reload*' or '*back*' button while processing a financial transaction. The presence of such features can be outright dangerous if a web application is used for controlling a medical system or a nuclear plant. In addition to predefined browser buttons, many of the predefined browser menu items – especially those that are displayed when right-clicking objects on the screen – are meaningless for applications. Web applications should preferably be able to override such features with application-specific behavior.

3.2. Networking and Security Issues

The document-oriented history of the web browser is apparent also when analyzing the restrictions and limitations that web browsers have in the area of networking and security. Many of these limitations date back to conventions that were established early on in the design of the browser. Furthermore, some of the restrictions are "folklore" and have never been fully documented or standardized.

The "Same Origin" networking policy is problematic. A central security-related limitation in the web browser is the "same origin policy" that was introduced originally in Netscape Navigator version 2.0. The philosophy behind the same origin policy is simple: it is not safe to trust content loaded from arbitrary web sites. When a document containing a script is downloaded from a certain web site, the script is allowed to access resources only from the same web site but not from other sites. In other words, the same origin policy prevents a document or script loaded from one web site ("origin") from getting or setting properties of a document from a different origin. The same origin policy has a number of implications for web application developers. For instance, a web application loaded from a web site cannot easily go and access data from other sites. This makes it difficult to build deploy web applications that combine content from multiple web sites. Special proxy arrangements are usually needed on the server side to allow networking requests to be passed on to external sites. Consequently, when deploying web applications, the application developer must be closely affiliated with the owner of the web server in order to make the arrangements for accessing the necessary sites from the application.

Only a limited number of simultaneous network requests allowed. Early on in the history of the Web another design convention was established that prevents a web browser from creating too many simultaneous HTTP requests. Such limitations were introduced to prevent too much web traffic from being created. Even today, most web browsers allow only a limited number (e.g., two or four) of network requests to be created simultaneously. With highly interactive web applications that require a lot of data from several sites asynchronously, such limitations can cause problems, especially if some of the sites do not respond to requests as quickly as expected.

No access to local resources or host platform capabilities. Web documents and scripts are usually run in a sandbox that places various restrictions on the resources and host platform capabilities that the web browser can access. For instance, access to local files on the machine in which the web browser is being run is not allowed, apart from reading and writing cookies. The sandbox security limitations prevent a malicious web site from altering the local files on the user's local disk, or from uploading files from the user's machine to another location. Unfortunately, the sandbox security limitations of the web browser make it difficult to build web applications that utilize local resources or host platform capabilities. Consequently, it has been nearly impossible to write web applications that would, e.g., be usable also in offline mode without an active network connection. These problems are gradually being solved with libraries such as WebDAV (Web-based Distributed Authoring and Versioning) [8] and Google Gears (http://gears.google.com/).

A more fine-grained security model is missing. The key point in all the limitations related to networking and security is that there is a need for a more fine-grained security model for web applications. On the Web today, applications are second-class citizens that are on the mercy of the classic, "one size fits all" sandbox security model of the web browser. This means that decisions about security are determined primarily by the site (origin) from which the application is loaded, and not by the specific needs of the application itself. Even though some interesting proposals have been made [9], currently there is no commonly accepted finer-grained security model for web applications or for the Web more generally.

3.3. Interoperability and Compatibility Issues

Incompatible browser implementations. A central problem in web application development today is browser incompatibility. Commercial web browsers have incompatibilities in various areas. For instance, the DOM implementations vary from one browser to another. DOM attribute names can vary from browser to browser; even seemingly trivial attributes such as window width and height have different names in different browsers. The JavaScript implementations have known differences, e.g., in the area of how event handlers can be triggered programmatically. The graphics libraries supported by the browsers have also been implemented differently. All these differences make it difficult to implement cross-platform, cross-browser web applications that would run identically on all browsers.

Disregard for official standards. Some browser vendors have a tendency to favor their own technologies in lieu of official World Wide Web Consortium (W3C) or ECMA standards. For instance, the JavaScript graphics libraries in the Lively Kernel depend on the W3C Scalable Vector Graphics (SVG) (http://www.w3.org/TR/SVG/) standard. Unfortunately, SVG support is not yet available in one of the commercially most important web browsers.

Lack of standards for important areas such as advanced networking, graphics or media. The Java programming language has exceptionally rich class libraries that have been standardized over the years using the Java Community Process^{SM} (http://www.jcp.org/). In contrast, during the development of the Lively Kernel we noticed that JavaScript libraries available for web application development are still surprisingly immature and incomplete. No widely accepted standards exist for areas such as advanced networking and graphics, audio, video and other advanced media capabilities. Although such libraries have been defined as part of external JavaScript library development activities, such as Dojo (http://www.dojotoolkit.org/), no officially accepted W3C or ECMA standards for these areas exist.

3.4. Development and Testing Issues

The power of the World Wide Web stems largely from the absence of static bindings. When a web site refers to another site or a resource such as a bitmap image, or when a JavaScript program accesses a certain function or DOM attribute, the references are resolved at runtime without static checking. It is this dynamic nature that enables the flexible combination of content from multiple web sites and, more generally, allows the Web to be "alive" and to evolve constantly with no central planning or control. The dynamic nature of the Web has various implications for application development and testing.

Evolutionary, stepwise development style is needed. For an application developer, the extreme dynamic nature of the Web poses new challenges, causing some fundamental changes in the development style. Basically, the development style needs to be based on stepwise refinement [10]. Such a style is closer to the "exploratory" programming used in the context of dynamic programming languages such as Smalltalk, Self or Lisp, rather than the style used with more static, widely used languages such as C, C++ or Java.

Completeness of applications is difficult to determine. Web applications are generally so dynamic that it is impossible to know statically, ahead of application execution, if all the structures that the program depends on will be available at runtime. While web browsers are designed to be error-tolerant and will ignore incomplete or missing elements, in some cases the absence of elements can lead to fatal problems that are impossible to detect before execution. Furthermore, with scripting languages such as JavaScript, the application can even modify itself on the fly, and there is no way to detect the possible errors resulting from such modifications ahead of execution. Consequently, web applications require significantly more testing (especially coverage testing) to make sure that all the possible application behaviors and paths of execution are tested comprehensively.

No support for static verification or static type checking. In the absence of well-defined interfaces and static type checking, the development style needed for web application development is fundamentally different from conventional software development. Since there is no way to detect during the development time whether all the necessary components are present or have the expected functionality, applications have to be written and tested piece by piece, rather than by writing tens of thousands of lines of code ahead of the first execution. Such piecemeal, stepwise development style is similar to the style used with programming languages that are specifically geared towards exploratory programming.

Incremental testing is required. Due to its highly permissive, error-tolerant nature, JavaScript programming requires an incremental, evolutionary approach to testing as well. Since errors are reported much later than usual, by the time an error is reported it is often surprisingly difficult to pinpoint the original location of the error. Error detection is made harder by the dynamic nature of JavaScript, for instance, by the option to change some of the system features on the fly. Furthermore, in the absence of strong, static typing, it is possible to execute a program and only at runtime realize that some parts of the program are missing. For all these reasons, the best way to write JavaScript programs is to proceed step by step, by writing (and immediately testing) each new piece of code. If such an incremental, evolutionary approach is not used, debugging and testing can become quite tedious even for relatively small JavaScript applications.

Code coverage testing is important. The dynamic, interactive nature of JavaScript makes testing deceptively easy. In the presence of an interactive command shell and the '*eval*' function, each piece of code can be run immediately after it has been written. Unfortunately, the use of such immediate testing approach does not guarantee the program to be bug-free or complete. In a static programming language, many simple errors will be caught already during the compilation of the program. In contrast, in a dynamic language, it is not possible to know statically if a piece of code that has never been executed will actually run without problems. As programs may contain numerous rarely executed branches (for instance, exception handlers) *code coverage testing* is very important. Still, even with 100% code coverage, it is possible that further problems will be found.

3.5. Deployment Issues

Anything that is made available on the World Wide Web is instantly accessible by anybody using a web browser. On the Web, there is no longer any need to do “shrink-wrapped” software releases. Even more importantly, the need for manual application installation or upgrades will go away. Ideally, the user will simply point the web browser to a site containing an application, and the latest version of the application will start running automatically. Release cycles will become considerably shorter. All these changes will be significant improvements compared to the traditional way of deploying desktop software. There are issues associated with such a deployment model, though, as discussed below.

Applications are “always on”. With the instant deployment model, applications are downloaded directly from the Web. The applications are “always on” in the sense that all the changes made to them will be immediately visible to all users who subsequently download the application. Since many of the users may still be using an earlier version, any updates to the application will have to be made carefully. For instance, if the application's internal data formats on a web server database change, backwards compatibility must be taken into account, since there may still be thousands of users who are using an older version of the application.

Towards “nano-releases”. A software release is the distribution of an initial or new and upgraded version of a computer software product. Traditionally, new software releases have occurred relatively infrequently, perhaps a few times per year for a major software product such as a word processor or spreadsheet application, or a few times per month for some business-critical applications in early stages of their deployment cycle. The instant deployment model will change all this, allowing new releases to be made much more frequently. In the ultimate scenario, a new release occurs each time changes are made to the system, perhaps even several times a minute. The possibility of such “*nano-releases*” has not been investigated much so far, but is bound to have significant long-term impacts in the software industry.

Perpetual beta syndrome. The transition towards web applications will make releases deceptively simple. When combined with the use of dynamic languages that allow incomplete software to be run, it becomes dramatically easier to release software in early stages of its development. This will lead to "*perpetual beta syndrome*": many software applications will never reach a point when they are actually ready for prime-time use. Only those applications and web sites that will become adequately popular will ever reach maturity while others will stay in beta form perpetually.

Fragmentation problems. The instant deployment model is closely related to the compatibility issues discussed earlier in Section 3.3. The instant deployment, "zero-installation" model works smoothly only as long as the target platform – in this case the web browser – is identical for all the users. If different browser versions or additional plug-in components are required, application distribution becomes considerably more challenging. From the application developer's viewpoint, this results in *fragmentation*: the need to build multiple versions of the same application for different platform variants. Such fragmentation problems are familiar from the mobile software industry, in which there are hundreds or thousands of different target devices (mobile phones), each with its own characteristics and peculiarities.

3.6. Performance Issues

Until recently, performance problems associated with web pages were more commonly associated with network latency and other connectivity issues, rather than with the performance of the web browser or the web page itself. However, now that people have started running real applications on the Web, performance problems have become apparent.

Inadequate JavaScript performance. Current JavaScript virtual machines are unnecessarily slow. Even though JavaScript is a significantly more dynamic language than, for instance, the Java programming language, there is no fundamental reason for JavaScript programs to run 10-100 times slower than comparable Java applications. At the very minimum, JavaScript virtual machine performance should be comparable to optimized Smalltalk virtual machine implementations, which is not yet the case. Fortunately, a number of higher-performance JavaScript virtual machines are on their way, including Mozilla's new virtual machine *Tamarin* (http://www.mozilla.org/projects/tamarin/).

Inadequate memory management capabilities. Current JavaScript virtual machines have simple, 1970's style garbage collectors and memory management algorithms that are poorly suited to large, long-running applications. For instance, with large applications that allocate tens of megabytes of memory, garbage collection pauses in the Mozilla SpiderMonkey JavaScript virtual machine (VM) (http://www.mozilla.org/js/spidermonkey/) can be excessively long, up to tens of seconds even on a fast machine. As in the VM performance area, with modern virtual machine implementation techniques memory management behavior could be improved substantially.

Inadequate graphics library performance. Application performance is typically a combination of many factors. The performance of the underlying execution engine, such as a JavaScript virtual machine, is in itself insufficient to guarantee the optimal performance of the application. Based on our experience, a major performance bottleneck in today's web browsers is graphics library performance. Graphics engines, such as the engines available for Scalable Vector Graphics (SVG), can be surprisingly slow. For highly interactive environments such as the Lively Kernel, this can have a significant negative impact on performance.

Inefficient bindings between the browser and other components. A great deal of the performance problems in the web application area can be attributed to inefficient communication between the browser and various other components. For instance, when the coordinates of a graphical object are passed from a JavaScript application to the browser (DOM) and ultimately to a native graphics library that draws the object, it is common to convert the numeric parameters into strings and then back to numbers again, possibly several times during the process. Such conversions can easily slow down graphics performance by an order of magnitude. In general, a lot of room for optimization remains in the area of native communication interfaces between the web browser, JavaScript engine and graphics libraries.

4. Solutions and Recommendations

Compared to how dramatically web usage has increased since the early 1990s, it is remarkable how little the web browser has changed since it was originally introduced. In general, web browsers are already so widely established that it may seem rather difficult to try to make any significant changes in the design or the behavior of the browser. However, given how quickly the use of web applications is increasing, it is quite possible that web browsers will have to adapt to accommodate a more application-oriented approach, in addition to the document-oriented approach that dominates the Web today.

Solving the usability and user interaction issues. The usability issues of the web browser seem relatively easy to fix. Basically, in order to support applications with direct manipulation and desktop-style user interaction, the I/O model of the web browser needs to be enhanced and complemented with capabilities familiar from the world of desktop applications. The problems in this area boil down to three basic issues: (1) the cumbersome I/O model of the web browser, (2) the presence of some browser features that are semantically problematic in the context of real applications, and (3) the absence of portable solutions for important user interaction features such as cut/copy/paste support. We have presented solutions to these issues in our earlier paper [5].

Solving the networking and security issues. The networking and security issues arise from the combination of the current "one size fits all" browser security model and the general document-oriented nature of the web browser. Decisions about security are determined primarily by the site (origin) from which the web document is loaded, not by the specific needs of the document or application. Such problems could be alleviated by introducing a more fine-grained security model, e.g., a model similar to the comprehensive security model of the Java SE platform [11] or the more lightweight, permission-based, certificate-based security model introduced by the MIDP 2.0 Specification for the Java™ Platform, Micro Edition (Java ME) [12].

The biggest challenges in this area are related to standardization, as it is difficult to define a security solution that would be satisfactory to everybody while retaining backwards compatibility. Also, any security model that depends on application signing and/or security certificates involves complicated business issues, e.g., related to who has the authority to issue security certificates. Therefore, it is likely that any resolutions in this area will still take years. Meanwhile, a large number of security groups and communities, including the Open Web Application Security Project (OWASP), the Web Application Security Consortium (WASC), and the W3C Web Security Context Working Group, are working on the problem.

Solving the interoperability and compatibility issues. As in the security area, the issues in the browser compatibility area are heavily dependent on standardization. In order to improve compatibility, an independently developed browser compatibility test suite, similar to the test suites available for the Java platform, would be very valuable. For each new browser feature, a reference implementation should also be made available. Having an independent third-party compatibility test organization might also help. If such an organization were available, new browser versions could be subjected to third-party compatibility testing before the new versions of the browser will be released to the public.

In general, improved communication and collaboration between the browser vendors are key to any improvements in this area. Additional standardization work is needed especially in the area of JavaScript library specification, where APIs are still missing from important areas such as advanced networking and graphics, audio, video and other advanced media capabilities.

Solving the development and testing issues. As we discussed earlier, the transition from conventional applications to web applications will result in a shift away from static programming languages such as C, C++ or C# towards dynamic programming languages such as JavaScript, PHP or Python. Since mainstream software developers are often unaware of the fundamental development style differences between static and dynamic programming languages, there is a need for education in this area. Developers need to be educated about the evolutionary, exploratory programming style associated with dynamic languages, as well as agile development methods and techniques that are available for facilitating such development.

In the testing area, there is an increased need for code coverage testing to ensure that all the parts of the applications are tested appropriately in the absence of static checking. Some of the problems can also be solved by tool support. For instance, static verification tools, such as *jslint* (http://www.jslint.com/), can be valuable in checking the integrity of an application before its actual execution.

Solving the deployment issues. One of the main benefits of the Web is instant worldwide deployment: Any artifact that is posted on the Web is immediately accessible to anybody in the world who has a web browser. This "instant gratification" dimension will revolutionize the deployment and distribution of software applications, and will imply various changes in the business model of almost everyone in the software industry.

One of the main challenges in the deployment area is to define a model that addresses the fundamental changes in the nature of applications that we discussed above: applications that are always on, the ever-shortening release cycles, and the perpetual beta syndrome. Detailed discussion on this issue falls beyond this paper, but we plan to focus on these topics in more detail in another research paper.

Solving the performance issues. As already mentioned, the JavaScript virtual machines, graphics library implementations, and native function bindings in today's web browsers are surprisingly slow. Now that people have started running significant desktop-style applications on the Web, these performance problems are becoming increasingly apparent. Fortunately, solutions in this area are relatively straightforward. Techniques for high-performance virtual machine implementation have been investigated for decades. Plenty of existing expertise exists in this area, both to support faster execution and more efficient memory management. To improve graphics performance and native function bindings, various techniques are also available, including closer integration with hardware-accelerated graphics engines.

5. Conclusions

For better or worse, the World Wide Web is increasingly the platform of choice for advanced software applications. Web-based applications require no installation or manual upgrades, and they can be deployed instantly worldwide. The transition towards web-based applications means that the web browser will become the primary target platform for software applications, displacing conventional operating systems and specific computing architectures and platforms from the central role that they used to have. As a consequence, software developers will increasingly write software for the Web rather than for a specific operating system or hardware architecture.

Web-based applications will open up entirely new possibilities for software development, and will ideally combine the best of both worlds: the excellent usability of conventional desktop applications and the enormous worldwide deployment potential of the World Wide Web. While the web browser is not an ideal platform for desktop-style applications, the instant deployment aspect makes web applications inherently superior to conventional desktop-style applications. With our own work on the Sun Labs Lively Kernel, we have demonstrated that there is a better way to build browser-based web applications that support rich user interaction, advanced graphics, integrated development and deployment, and online collaboration. We hope that such features will become commonplace in web application development in the near future.

6. References

[1] Goodman, D., *Dynamic HTML: The Definitive Reference*. O'Reilly Media, 2006.

[2] Flanagan, D., *JavaScript: The Definitive Guide*, 5th Edition. O'Reilly Media, 2006.

[3] Paulson, L.D., Developers shift to dynamic programming languages, *IEEE Computer*, Vol 40, nr 2, February 2007, pp. 12-15.

[4] Crane, D., Pascarello, E, James, D., *Ajax in Action*. Manning Publications, 2005.

[5] Mikkonen, T., Taivalsaari, A., Web Applications: Spaghetti code for the 21st century. Technical Report TR-2007-166, Sun Microsystems Laboratories, 2007.

[6] Maloney, J.H., Smith, R.B., Directness and Liveness in the Morphic User Interface Construction Environment. Proceedings of the 8th annual ACM Symposium on User Interface and Software Technology (UIST), Pittsburgh, Pennsylvania, 1995, pp. 21-28.

[7] Mikkonen, T., Taivalsaari, A., Using JavaScript as a Real Programming Language. Technical Report TR-2007-168, Sun Microsystems Laboratories, 2007.

[8] Dussealt, L., *WebDAV: Next-Generation Collaborative Web Authoring*. Prentice-Hall Series in Computer Networking and Security, 2003.

[9] Yoshihama, S., Uramoto, N., Makino, S., Ishida, A., Kawanaka, S., De Keukelaere, F., Security Model for the Client-Side Web Application Environments. IBM Tokyo Research Laboratory presentation, May 24, 2007.

[10] Wirth, N., Program development by stepwise refinement. Communications of the ACM vol 14, nr 4 (Apr) 1971, pp. 221-227.

[11] Gong, L., Ellison, G., Dageforde, M., *Inside Java™ 2 Platform Security: Architecture, API Design, and Implementation*, 2nd Edition. Addison-Wesley (Java Series), 2003.

[12] Riggs, R., Taivalsaari, A., Van Peursem, J., Huopaniemi, J., Patel, M., Uotila, A., *Programming Wireless Devices with the Java™ 2 Platform, Micro Edition* (2nd Edition). Addison-Wesley (Java Series), 2003.