

# **Towards a Uniform Web Application Platform for Desktop Computers and Mobile Devices**

**Tommi Mikkonen and Antero Taivalsaari**



# Towards a Uniform Web Application Platform for Desktop Computers and Mobile Devices

Tommi Mikkonen and Antero Taivalsaari

SMLI TR-2008-177

October 2008

## Abstract:

Two parallel emerging trends are currently taking place in the software industry. First, software applications are moving to the World Wide Web: as part of the Web 2.0 transition, web pages are becoming increasingly interactive and will allow the use of desktop-style applications on the Web. Second, mobile devices are becoming an important platform for web applications: the devices are equipped with ever faster processors and network connections, allowing web applications to run on mobile devices as well. Although various usability differences will likely remain, we believe that in the long run there will be a uniform web application platform (“One Web”) that can be used with different types of terminals, including desktop computers and mobile devices.

In this paper, we summarize our experiences in porting the Sun™ Labs Lively Kernel, an exceptionally interactive web programming environment developed at Sun Microsystems Laboratories, onto a Nokia N810 mobile device. We report our experiences based on two different approaches that were used. First, we ported the system onto a regular web browser running in the mobile device. Second, we developed a custom-built native execution environment that provides more direct and extensive access to the underlying resources of the system. Based on these experiments, we will discuss the lessons learned as well as provide directions and guidance for future work.



Sun Labs  
16 Network Circle  
Menlo Park, CA 94025

## email addresses:

tommi.mikkonen@sun.com  
antero.taivalsaari@sun.com

© 2008 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

#### TRADEMARKS

Sun, Sun Microsystems, the Sun logo, Java ME, JavaScript, Java Platform, Micro Edition, and Sun Labs Lively Kernel are trademarks or registered trademarks of Sun Microsystems, Inc. or its subsidiaries in the United States and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <[jeanie.treichel@sun.com](mailto:jeanie.treichel@sun.com)>. All technical reports are available online on our website, <http://research.sun.com/techrep/>.

# Towards A Uniform Web Application Platform for Desktop Computers and Mobile Devices

Tommi Mikkonen    Antero Taivalsaari  
[tommi.mikkonen@sun.com](mailto:tommi.mikkonen@sun.com)    [antero.taivalsaari@sun.com](mailto:antero.taivalsaari@sun.com)

Sun Microsystems Laboratories  
P.O. Box 553 (TUT)  
FI-33101, Tampere, Finland

## 1. Introduction

The software industry is currently in the middle of a paradigm shift towards web-based software. As part of the ongoing Web 2.0 transition, web pages are becoming increasingly interactive and will allow the use of desktop-style application on the Web. Examples of such applications include the Google Docs online word processing and spreadsheet applications (<http://docs.google.com/>), as well as various web-based instant messaging systems and games. In the new era of web-based software, applications live on the Web as services that can be accessed with a web browser. The services consist of data, code and other resources that can be located anywhere in the world. The services and applications require no installation or manual upgrades; this makes the deployment of applications exceptionally rapid and simple. Ideally, applications should also support user collaboration and social interaction, i.e., allow multiple users to interact and share the same applications and data over the Internet.

There is another parallel transition currently occurring in the software industry: mobile devices are becoming an important application platform and a gateway to the Web. While the Web has conventionally been accessed from a personal computer, the increasing CPU speeds, memory capacity, network bandwidth and “all-you-can-eat” monthly network service plans are rapidly making mobile web usage and mobile software applications more practical.

These two transitions – the movement towards web-based software and towards web-enabled mobile devices – are transforming the software industry in many important ways. We believe that in the long run the popularity of the Web will make it *the* most important software application platform in the world. We also believe that the sheer popularity of mobile devices – there are already nearly three billion mobile device users today – will eventually lead us to a common web application platform that can be used with different types of terminals, including desktop computers and mobile devices.

In this paper, we summarize our experiences in porting the Sun™ Labs Lively Kernel – an exceptionally interactive web programming environment developed at Sun Microsystems Laboratories – onto a Nokia N810 mobile device. We report our experiences based on two different approaches that were used. First, we ported the system onto a regular web browser running in the mobile device. Second, we developed a custom-built native execution environment that provides more direct and extensive access to the underlying resources of the

system. Based on these experiments, we discuss the lessons learned as well as provide directions and guidance for future work.

The structure of this paper is as follows. In Section 2 we provide some background on mobile web applications. In Section 3, we introduce the Sun Labs Lively Kernel, a flexible web programming environment designed and implemented at Sun Microsystems Laboratories. In Sections 4 and 5 we summarize our experiences in porting the Lively Kernel onto a mobile device, and provide insights into the various implementation technologies that we have used. In Section 6 we discuss the lessons that we have learned during the porting efforts, and finally, Section 7 concludes the paper.

## **2. Towards Mobile Web Applications**

The World Wide Web is bound to be the most important media in the 21<sup>st</sup> century. Basically, anything that can be digitalized and shared with others online – photos, music, videos, maps, documents, news, real estate data, and so on – is already on its way to the Web. Very likely, the Web will also become the most popular platform for software applications.

It has been estimated that there will be three billion mobile device users in the world by 2009, versus about one billion personal computer users. Given that mobile devices are becoming web-enabled, and the fact that in many countries (especially in developing regions) a mobile device is often the only means for people to access the Web, mobile web applications will inevitably become very important in the future.

Currently there are still major obstacles to the widespread use of mobile web applications, including network bandwidth or cost issues, CPU performance problems, memory capacity, and screen size limitations. Therefore, many web services and applications that run well in a desktop browser are next to unusable in a mobile device, even if the technical capabilities needed for executing the applications (such as a mobile web browser with the necessary plug-in components and libraries) were available in the mobile device.

Over the years, numerous solutions have been proposed to overcome the challenges above, including custom solutions to create the “mobile web”, i.e., custom-built web technologies intended specifically for mobile devices. Such technologies include the infamous Wireless Application Protocol (WAP) (<http://www.wapforum.org/>), or various special gateways that automatically transliterate and customize web services and content for use in mobile devices with different screen sizes. Many popular web sites offer special versions of their services and applications that have been tailored to mobile devices. Examples of systems with available special mobile versions include Google Maps (<http://maps.google.com>), Google Mail (<http://www.gmail.com/>), and the Marketwatch.com financial data service (<http://www.marketwatch.com/>). Furthermore, a special internet domain “.mobi” has been created specifically for mobile services (see, for instance, <http://www.nokia.mobi/>).

Unfortunately, the development of custom-built mobile web services and applications requires additional engineering effort, which leads to added cost in the design, implementation, testing, maintenance, and deployment of the services. In practice, the effort needed for maintaining a high-quality service that is usable with all the desktop browsers is already challenging enough.

This means that web service providers often have less interest in supporting those services that are specific to mobile devices. Consequently, many web services that have customized versions for mobile devices are incomplete or poorly maintained. As the number of mobile web users increases, the situation will probably improve.

In the long run, we believe that there will be “One Web”, with content, services and applications that can be used from any type of terminal, including desktop computers and mobile devices, irrespective of the target environment [MVB08]. While problems related to small screen sizes and usability will remain, the other problems arising from CPU speed differences, memory capacity differences, and network bandwidth limitations are likely to disappear over time. Eventually it will be possible to use the same web applications and services in desktop computers and other types of client devices. This trend is currently strengthened by the recent emergence of “netbooks” and “sub-PCs”: very small, no-frills, low-cost laptop PCs that can be used for web browsing and not much else.

In the remainder of this paper we will summarize our experiences in building and porting a web application environment that is intended to support the “One Web” principle, i.e., support applications that range from desktop systems to mobile devices. The application environment is intended to be usable in a regular web browser in a desktop environment as well as in mobile devices. However, we have also built a more optimized custom implementation that runs natively for improved performance and usability. In Section 3, we will provide a general introduction to the system that we have built. The browser-based implementation is described in Section 4, and the custom-built native implementation will be described in Section 5. The discussion in this paper is limited to those parts of the system that run on the client side in the mobile device. The software associated with the web server falls beyond the scope of this paper, except when it has particular relevance to the client side.

### 3. Sun Labs Lively Kernel

At Sun Labs, our team has developed a new web programming environment called the *Sun Labs Lively Kernel* (<http://research.sun.com/projects/lively>) [TMI08, IPU08]. The Lively Kernel is an exceptionally interactive, “zero-installation” web application development platform that has been written entirely in the JavaScript™ language. The Lively Kernel runs in an ordinary web browser without installation or any browser plug-in components whatsoever; it supports desktop-style applications with rich user interface features and direct manipulation capabilities; it enables application development and deployment in a web browser with no installation or upgrades, using nothing more than existing web technologies. In addition to its application execution capabilities, 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.

A screen snapshot of the Lively Kernel is presented in Figure 1. Figure 1 shows the desktop version of the Lively Kernel running in the Safari 3.0 web browser. A number of applications and tools are displayed, including a clock widget, a map application, a rotating Sun 3D logo, a JavaScript class browser and an object hierarchy browser. A link to another world (a synonym for web pages in our system), represented as a circular icon, is also visible.

Each visible object on the screen can be rotated, scaled and grabbed by the user, moved about by simply dragging it with mouse, and composed (“glued”) with other objects to form more complex graphical objects. All the objects can have associated timers for events that occur periodically. This mechanism can be used for animation effects such as automatic rotation, or automatic movement of objects, for instance.

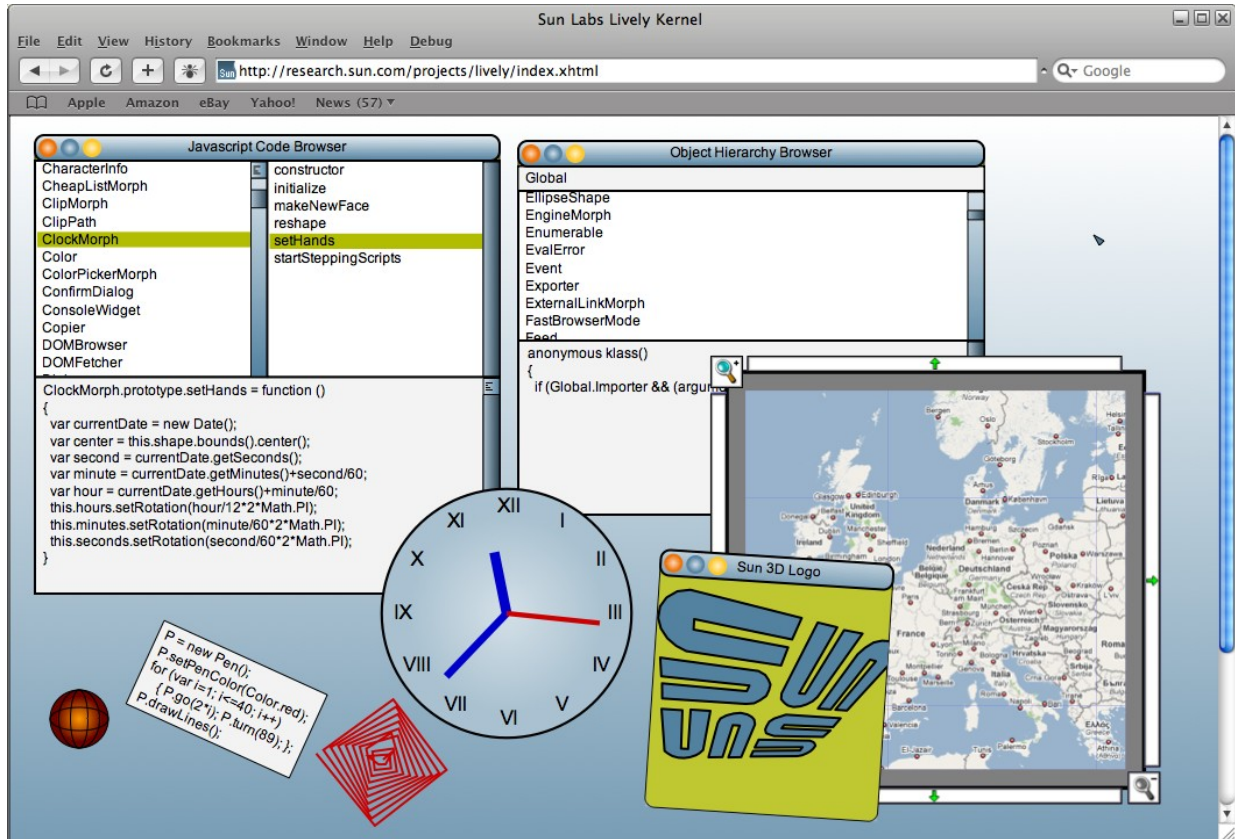


Figure 1: The desktop version of the Sun Labs Lively Kernel running in the Safari web browser

From the end user's viewpoint, the Lively Kernel behaves as shown in Figure 2. First, the user launches the Lively Kernel by typing a URL in a web browser. When the user presses the the return key to load the web page, the files that define the Lively Kernel system itself and the application(s) are downloaded from the web server for execution in the web browser (see 1(a) and 1(b) in Figure 2).

Note that no pre-installation or plug-in components are required for the Lively Kernel itself or for the applications running in the system, apart from the JavaScript engine and the graphics libraries that already exist in the browser. Once the Lively Kernel is running, the application will communicate with the web server using asynchronous HTTP (the XMLHttpRequest feature supported by all the modern web browsers). The Lively Kernel and the application(s) will continue running in the browser until the user closes the web browser or moves to another web page.

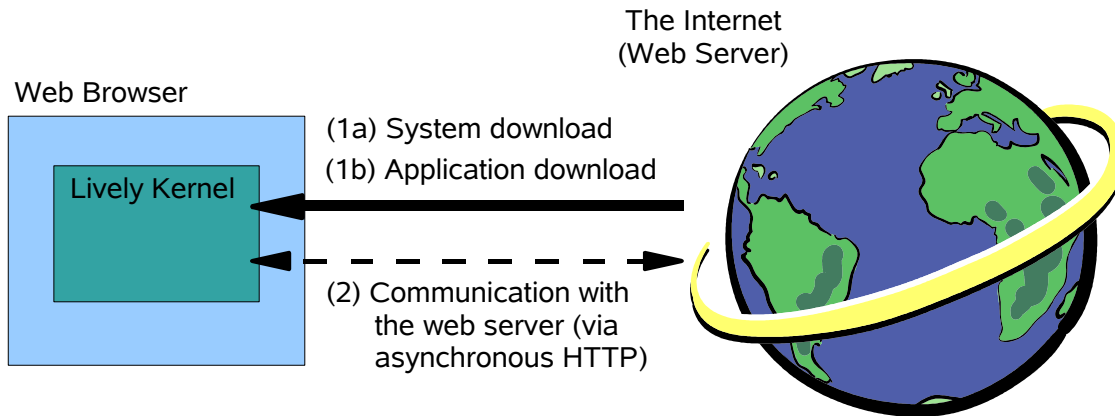


Figure 2: The high-level behavior of the Lively Kernel

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

- *JavaScript programming language.* The JavaScript programming language, a language supported by all the commercial web browsers today, is used as a fundamental building block for the rest of the system. Apart from small fragments of HTML and XHTML, most of the code of the Lively Kernel system is written in JavaScript.
- *Asynchronous HTTP networking.* All the networking operations in the Lively Kernel are performed asynchronously, utilizing the XMLHttpRequest feature familiar from Ajax [CPJ05]. 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.
- *Morphic user interface framework and widgets.* The Lively Kernel is built around a rich user interface framework called *Morphic* [Mal95, MaS95]. Morphic supports composable graphical objects, along with the machinery required to display and animate these objects, handle user inputs, and manage underlying system resources such as displays, fonts and color maps. Sophisticated built-in mechanisms are provided for object scaling, rotation, object style editing, as well as for defining user interface themes. In our Lively Kernel implementation, 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.
- *Built-in tools for developing, modifying and deploying applications on the fly.* The Lively Kernel includes tools – such as a class browser, object inspector, object hierarchy browser and a call stack viewer – that can be used for developing, modifying and deploying applications from within the Lively Kernel system itself [TMI08]. 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. The features make it possible, e.g., to write new JavaScript classes, modify or delete existing methods, or change the values of any attribute in the system. Such facilities are familiar from other dynamic programming environments such as Squeak [IKM97] and Self [UnS87].

A key difference between the Lively Kernel and other systems in the same area is our focus on *uniformity*. Our goal was 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 the goal was to do as much as possible using a single technology: JavaScript [Fla06]. However, the Lively Kernel also leverages 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 regular web browser without any plug-in components or external tools whatsoever.

In general, the Lively Kernel is intended to be an “*Open Web*” environment: based open standards, open source code, and content formats that are not controlled by a single vendor [Eic07]. Our focus on building a system that does not require any plug-ins or other components differentiates it from those systems that extend the browser functionality with additional, often proprietary components. Examples of such web application environments include Adobe Integrated Runtime (AIR) [FPM08] and Microsoft Silverlight [Mor08].

Given these capabilities and the fact that the system can run in an ordinary web browser, the Lively Kernel is an ideal research vehicle for studying the capabilities of the web browser as an application platform. The Lively Kernel has also served as a testbed for studying the use of the JavaScript language as a general-purpose programming language [MiT07].

## **4. Porting the Lively Kernel onto a Web Browser**

In this section we describe our experiences in porting the Lively Kernel system from a desktop browser onto a web browser running in a mobile device. In an ideal situation – assuming that web browsers for desktop computers and mobile devices were fully compatible – the mobile port of the Lively Kernel should not require any additional work at all. Unfortunately, the browsers are not yet fully interoperable and therefore additional work is required. We start the discussion by first providing a description of the desktop implementation of the Lively Kernel. After that, we summarize our experiences in porting it onto a web browser in a mobile device.

### *4.1 Desktop implementation of the Lively Kernel*

Since the Lively Kernel is intended to run in a regular web browser without plug-in components or any non-standard components, the implementation is heavily dependent on those technologies that are available in the web browser. Figure 3 provides a block structure illustration of the components of the Lively Kernel at runtime. The items within the solid black line represent components that are provided by the web browser, while the rest of the components are usually downloaded from the Web dynamically.

At the bottom of the diagram is the *Scalable Vector Graphics* (SVG) graphics library provided by the web browser (see more detailed description below). The SVG layer is not visible to Lively Kernel application developers, except in terms of the functionality that it provides. Next, we have the JavaScript virtual machine and the JavaScript core libraries [Fla06]. On top of the

JavaScript environment we use a JavaScript class library called *Prototype* that provides additional structure and utility functions to JavaScript applications. The Lively Kernel libraries and tools (including the Morphic user interface framework) then reside on top of the JavaScript environment and the Prototype library.

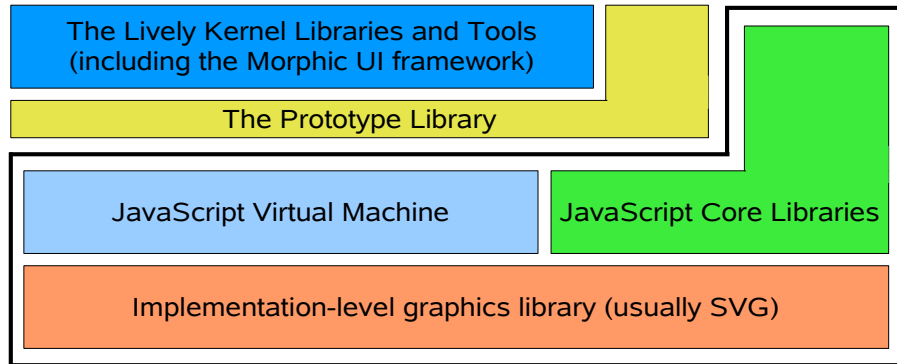


Figure 3. Components of the browser-based implementation of the Lively Kernel

As depicted in Figure 3, the Lively Kernel depends on a number of external libraries. These libraries are summarized in more detail below.

- **Scalable Vector Graphics (SVG).** Our current Lively Kernel implementation depends on an SVG engine that provides the necessary implementation-level support for our 2D graphics architecture, including the affine transformations and matrix operations needed for rotating and scaling objects. SVG is a declarative graphics language and specification designed by the World Wide Web Consortium (W3C) [SVG03]. It is supported by most web browsers, although the implementations are still somewhat incompatible or missing altogether from some web browsers. While SVG is commonly used via its XML-based declarative syntax, the SVG functionality of a web browser can also be accessed programmatically from JavaScript. The Lively Kernel is built on top of the programmatic JavaScript SVG interface. For further information on SVG, refer to the SVG 1.1 Specification:

<http://www.w3.org/TR/2003/REC-SVG11-20030114/>

Note that in the context of the Lively Kernel, SVG is treated as an *implementation-level API*. We have chosen to use SVG because of its functionality and widespread availability in commercial web browsers. However, to application developers, the Lively Kernel is an environment that is based purely on JavaScript, not SVG. Future releases of the Lively Kernel may include back-ends also for other graphics standards than SVG. An interesting alternative would have been the *Canvas API*, which is supported by some browsers and defined in Section 4.7.11 of the HTML 5 Specification [HTM08]. The restrictions of the Canvas API, especially those associated with text rendering, led us to abandon this alternative.

- **Prototype library.** The Lively Kernel uses a JavaScript class library called *Prototype* (see <http://www.prototypejs.org/>). The Prototype library provides convenient syntactic mechanisms for defining and extending JavaScript classes. Such mechanisms are missing from the widely used versions of the JavaScript (ECMAScript) language. In addition, the

Prototype library introduces an interface for using asynchronous HTTP networking in a browser-independent, platform-independent fashion. This is important since the implementation of the XMLHttpRequest interface still varies from one web browser to another. The Prototype library also provides some convenience functions for using collections such as arrays more effectively.

Figure 4 contains another block diagram illustration of the desktop implementation of the Lively Kernel. Basically, items inside the gray box labeled “Browser” are provided by the web browser. The specific items that we use in the Lively Kernel include the JavaScript engine (and the associated JavaScript core libraries), the SVG engine (for graphics rendering), and asynchronous networking (to prevent the UI interface from being blocked during network requests). The box labeled “Lively Kernel” represents those components (including the Morpheic framework) that are downloaded into the web browser when the Lively Kernel starts.

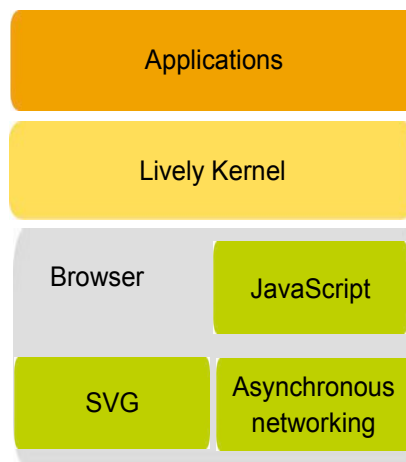


Figure 4. Components of the browser-based implementation of the Lively Kernel

In addition to the technologies listed above, the Lively Kernel relies on a minimal amount of HTML or XHTML code that will bootstrap the system when the Lively Kernel is launched from the web browser. This code loads the necessary JavaScript files, as well as initializes the SVG drawing area for use by the Lively Kernel.

Note that in its present form the Lively Kernel is not universally compatible with all the commercial web browsers. In particular, the Lively Kernel does not currently run in the Microsoft Internet Explorer because Internet Explorer supports a Microsoft-specific Vector Markup Language (VML) [VML98] instead of SVG. Internet Explorer does not provide support for XHTML either. However, we have recently built an implementation of the Lively Kernel for Microsoft Internet Explorer utilizing the Renesis SVG player plug-in by a company called Examotion (<http://www.examotion.com/>).

The Lively Kernel also has known minor incompatibilities with other web browsers such as Opera and Mozilla Firefox 2. The web browsers that currently run the Lively Kernel best are Safari 3 and Mozilla Firefox 3.

## 4.2 Porting the desktop implementation onto a mobile device

As a target device for our mobile port, we originally chose the *Nokia 770* device (<http://www.nokia.com/770>). The Nokia 770 device (and its successors *Nokia N800* and *Nokia N810*) is an internet tablet device, with a high-resolution (800 by 480 pixel) color display and a touch screen. A small number of physical buttons are provided for zooming and navigation, and for switching between regular and full-screen mode. The original 770 device was powered by a TI OMAP 1710 processor running at 220 MHz, while the more recent N810 device uses a TI OMAP 2420 processor running at 400 MHz. The N810 device has 128 megabytes of DRAM and 256 megabytes of built-in Flash memory. Unlike its predecessors, the Nokia N810 (<http://www.nokia.com/n810>) device also has an integrated QWERTY keyboard that can be used when necessary by sliding it open from under the device. As system software, all these Nokia devices use the Maemo Linux platform (<http://maemo.org/>), which is based on the Debian Linux distribution. In addition, the devices have some Nokia-specific software components, including a special widget set referred to as Hildon, which has been derived from GTK widgets [Gri00].

As can be seen from the specifications above, the Nokia internet table devices provide abundant processing power and memory for applications. Compared to desktop computers, performance is still modest, though, especially with respect to graphics performance.



Figure 5. The Nokia N810 device used in our porting activities

The original versions of the Maemo platform used Opera (<http://www.opera.com/>) as the web browser. However, starting from the OS2008 version of the Maemo platform, Mozilla Firefox (<http://www.mozilla.org/>) is used. Unfortunately, neither of these browsers for the Maemo

platform provides SVG support by default. For our porting activities, a custom-built SVG-enabled version of the Mozilla browser for the Maemo platform was made available to us. When the SVG-enabled browser was installed in the device, we started experimenting with the Lively Kernel in the device.

**Technical issues encountered during the initial porting effort.** During this experimentation phase, we ran into a number of problems. Even though the desktop and mobile versions of the same web browser are intended to be compatible, the mobile version of Firefox turned out to be far less permissive and error-tolerant than the desktop version. In other words, whereas the desktop version was able to tolerate and ignore minor errors and warnings, and successfully recover from more serious errors, the mobile version was far more likely to halt application execution or crash. This behavior, in combination with the very limited debugging capabilities available in the mobile device and in the web browser, made the initial porting effort rather strenuous.

Also, it turned out that some browser features, such as the support for SVG path objects in the Document Object Model (DOM) implementation, were not supported in the mobile version. There were also incompatibilities in the behavior of the underlying drawing canvas structure, as well as in event handling behavior. For instance, in a pen-based device it is not possible to move the mouse without touching the screen with the stylus, and therefore `mousemove` events are always accompanied with `mousedown` events (see the discussion about mouse events below.)

After dealing with these and some other minor incompatibilities, we had a subset of the Lively Kernel functionality and applications successfully running in the N810 device, including some archetypical Lively Kernel widgets such as the ClockMorph clock widget.

**Mismatches in the design of the Lively Kernel and the target device.** During the porting effort – in addition to the minor technical issues discussed above – we ran into more complex issues that were related mostly to the physical characteristics of the target device, especially screen size, mouse event handling and keyboard event handling. These topics are discussed in more detail below:

- *Screen size.* An obvious source of difficulties in our experiment was screen size. Even though the Nokia N810 device has an exceptionally large, high-resolution (800x480 pixel) display for a mobile device, the display is still small compared to most personal computers. Since the small screen could not easily accommodate several widgets or applications simultaneously, as is typical in the desktop version of the Lively Kernel, we ended up allocating the entire screen to individual applications and displaying a more limited set of widgets than in the desktop version. In order to utilize the limited screen size more efficiently, some UI features, such as window title bars, were eliminated or reduced in size.
- *Mouse events.* The Morphic graphics framework underlying the Lively Kernel was initially designed to run in a classic WIMP (Windows, Icons, Menus and a Pointing Device) target environment. While the Nokia N810 device has all these classic elements in place, including a pointing device (stylus), the user interaction model in a stylus-based device differs considerably from a computer with a conventional mouse. One critical difference is that in a stylus-based user interface it is not possible to move the pointer without touching the screen, i.e., to distinguish between ordinary `mousemove` events from mouse *drag* events (= those

mousemove events during which the mouse button is held down). This feature caused us considerable headache, since the Morphic framework has a number of functions that are performed when the mouse is moved but not dragged. In the absence of a keyboard, it was not easy to emulate the difference between mouse moves and drags, e.g., by using a command, option or shift key.

- *Keyboard events.* The Nokia 770 device that we used during our initial porting efforts does not have a keyboard at all. Rather, when text entry is required, the user is expected to use a virtual keyboard (or a handwriting recognizer) that is opened automatically by the Maemo system whenever the native text editing widgets of the Maemo platform are used. The limited keyboard support posed two major problems in porting the Lively Kernel.
  - 1) First, since only a virtual keyboard was available (which, when opened, would consume a lot of precious screen real estate), we could not easily use a number of operations in the Lively Kernel that rely on the presence of helper keys such as the command, option or shift key. For instance, object rotation and scaling in the desktop version of the Lively Kernel are commonly performed using these helper keys while dragging the corner handles of an object.
  - 2) The second major problem was that the virtual keyboard could not easily work with the text editing capabilities of the Lively Kernel. The virtual keyboard in the Maemo platform is closely associated with native widgets that require text entry, and there is no easy way to access the virtual keyboard functionality programmatically from JavaScript. Since the Lively Kernel implements most of the text editing capabilities in JavaScript, the system would have required keyboard events from the operating system even when no native text editing widgets were active. A lot of effort was spent on circumventing these problems.

The N810 device, with its integrated QWERTY keyboard, alleviated the problems above considerably. However, the use of some Lively Kernel functions, such as object rotation and scaling, is still clumsy because it is not easy to press multiple keys simultaneously while using a stylus on a handheld device without potentially dropping the device onto the floor.

**Performance issues.** By far the biggest challenges that we experienced with the web browser based mobile version of the Lively Kernel were related to performance. In short, the performance of the SVG-enabled browser based version could be best described as “glacial”, i.e., unacceptably slow.

We have analyzed performance problems related to JavaScript applications in more detail in another paper [TMI08]. As summarized in that paper, the performance issues can be grouped broadly in three categories: (1) the JavaScript virtual machines are still unnecessarily slow, (2) the graphics libraries in the web browsers are still slow, and (3) the bindings between the various components in the web browser are slow, because the internal communication occurs primarily through the Document Object Model, a structure not designed for high performance.

The performance problems in the browser-based mobile port of the Lively Kernel manifested themselves especially in the following two areas:

- *Startup time.* The initial application startup, including a number of relatively simple widgets that are loaded from the local file system of the mobile device, takes about one minute to perform. As the loading advances, more and more items keep popping up on the screen gradually, making it easy to monitor the startup progress. Based on our analysis, this “startup inertia” seems to be caused primarily by the time that it takes to create all the necessary internal SVG elements and DOM structures inside the web browser. While it is difficult to analyze the problems further without digging deeper into the internals of the web browser, the likely culprits are the memory management capabilities inside the browser and in the underlying operating system.
- *Event handling performance.* The Lively Kernel manages its own events. In essence, the web browser converts each native event (such as a `mousemove` or a `keydown` event) received from the operating system into a web browser DOM Event object. This DOM Event object is then translated by the Lively Kernel into a “synthetic” Lively Kernel event object. Each event object is processed by the Lively Kernel event handler routines that have been written in JavaScript. The JavaScript code includes steps such as mapping the mouse coordinates to a particular object, determining whether the user is performing a selection, or grabbing or moving an object, and so on, and eventually executing the associated application-defined event handling routine (also written in JavaScript). As a result of the event handling inertia inside the web browser and in the Lively Kernel, the system is often unable to respond to native events generated by the device quickly enough, generating a backlog of unprocessed events. This makes it difficult, e.g., to perform a drag-and-drop operation in a controlled fashion, and forms a major usability problem even in relatively simple applications. Since a large number of short-lived event objects are created, this approach also places a significant burden on the garbage collector of the JavaScript virtual machine.

Due to the difficulties summarized above, it became obvious that an alternative implementation was necessary if we wanted to have a practical mobile Lively Kernel implementation available before the more general browser-related and JavaScript-related problems would be solved in the mobile space. Therefore, we decided to build a another implementation that would eschew the full-blown web browser and construct a corresponding environment natively, with specific focus on improved JavaScript performance and graphics. That implementation will be described in the section below.

## **5. Native Mobile Implementation of the Lively Kernel (“ScriptBrowser”)**

In addition to the web browser based Lively Kernel port, we have also created a custom-built mobile version of the Lively Kernel that leverages the native graphics and networking APIs of the underlying operating system. For historical reasons, this system is known as the *ScriptBrowser*, for “a browser that is capable of executing and rendering scripts only”.

The ScriptBrowser resembles an ordinary web browser in that it allows the user to specify to an arbitrary URL in order to access a web site (see the screen snapshot in Figure 6). However, instead of downloading and rendering HTML pages, the ScriptBrowser is able to download and execute JavaScript code and render graphics using the built-in graphics APIs only. Conventional

web page formats such as HTML and CSS are not supported. Graphics are rendered by mapping the graphics calls from JavaScript to the underlying graphics API, instead of communicating indirectly (and inefficiently) via the Document Object Model (DOM).

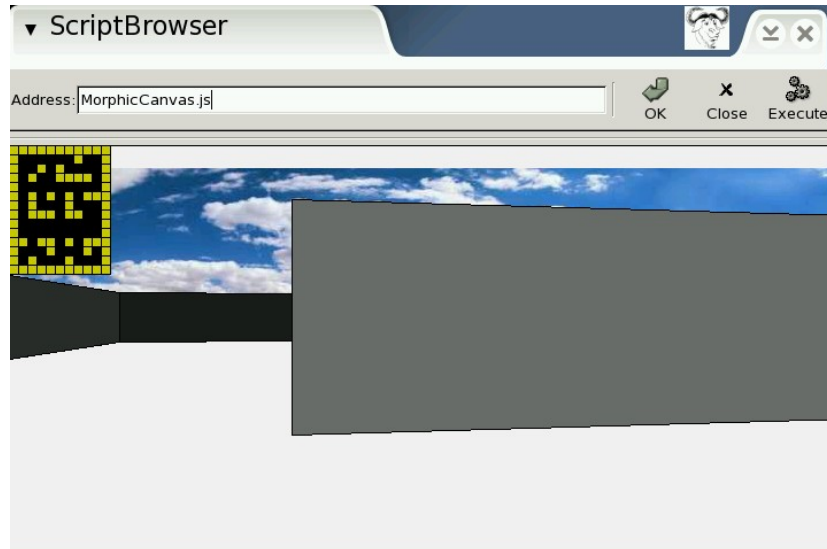


Figure 6. ScriptBrowser: A native, mobile implementation of the Lively Kernel running a game application (screen snapshot from the N810 emulator)

In general, the ScriptBrowser does not support the conventional DOM at all. Rather, applications are expected to maintain their internal state explicitly and perform graphics calls directly just like conventional (desktop) software applications usually do. This allows us to optimize performance and avoid most of the performance and startup time issues that hamper the conventional web browser environment [TMI08]. For instance, application startup in the ScriptBrowser is generally about 6-10 times faster than in the corresponding web browser based mobile version of the Lively Kernel.

The ScriptBrowser environment is built around the *SpiderMonkey* JavaScript virtual machine (<http://developer.mozilla.org/en/docs/SpiderMonkey>) developed by Mozilla (originally by Netscape). SpiderMonkey is implemented in the C programming language, and it provides a simple yet powerful porting interface for embedding the JavaScript engine into a target platform and specifying which native functions are then made available to JavaScript applications. Using SpiderMonkey's porting interface we added support for graphics and asynchronous networking on top of the native APIs provided by the underlying operating system. This resulted in the architecture depicted in Figure 7.

The block diagram in Figure 7 is very similar to the diagram illustrating the browser-based implementation of the Lively Kernel in Figure 4. However, instead of relying on a graphics engine provided by a web browser, the ScriptBrowser can map the JavaScript graphics calls directly onto the underlying native graphics APIs. Correspondingly, asynchronous networking calls are mapped onto the networking facilities provided by the underlying operating system. An additional essential difference, not depicted in Figure 7, is the absence of the DOM in the ScriptBrowser.

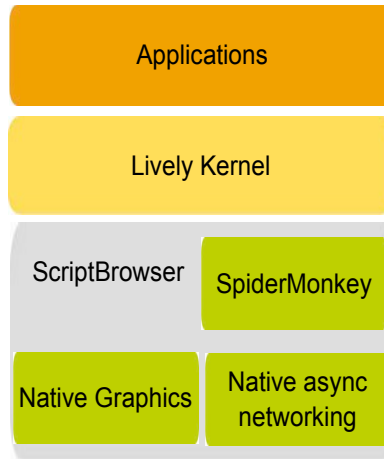


Figure 7. Components of the native version the Lively Kernel (ScriptBrowser)

In the following subsections, we will discuss the various aspects of the ScriptBrowser environment in more detail, starting from application loading and startup, and proceeding towards topics such as graphics, event handling, networking and security.

### 5.1 Application loading and startup

Application loading in the ScriptBrowser resembles an ordinary web browser in the sense that the user types a URL in the address bar (see Figure 6) in order to open a new page (in this case: a new application). However, since the ScriptBrowser is able to recognize JavaScript code only, the file located at the URL must contain executable JavaScript source code, not HTML.

In a normal web browser, the names of the files containing executable scripts are usually specified in an HTML file. To support the splitting of the application into multiple source files in the absence of HTML support, a special *AsyncHttpRequest* primitive has been added to the JavaScript engine inside the ScriptBrowser so that a JavaScript source code file (application) can initiate the downloading and evaluation of additional JavaScript files. This *AsyncHttpRequest* primitive allows a JavaScript application to load additional JavaScript source code files incrementally, in the same fashion as the *XMLHttpRequest* operation familiar from Ajax allows the downloading of web pages and other data on the background. The *AsyncHttpRequest* operation and its benefits are described in more detail in another paper.

In the ScriptBrowser, JavaScript files can be loaded from both from the local file system and from the Web. If a file is downloaded from the Web, it is first downloaded to the local file system; when the download is complete, the file is read and evaluated by the JavaScript virtual machine. During the loading process, the JavaScript code explicitly initializes all the variables and data structures that it needs. As already mentioned before, no DOM support is provided.

Loading and initializing only those data structures that are associated with the JavaScript application has a significant positive impact on application startup time. Generally speaking, applications in the ScriptBrowser are loaded 6-10 times faster than in the browser-based mobile version of the Lively Kernel. In a typical situation, the loading of a JavaScript application in the

ScriptBrowser takes about 6-10 seconds instead of over a minute in the corresponding web browser based mobile version of the Lively Kernel.

Since we have not been able to instrument and debug the internals of the web browser used in the N810 mobile device, we have not been able to pinpoint and analyze the precise reasons for the performance differences.<sup>1</sup> However, it seems that the majority of the startup time in the web browser based mobile version of the Lively Kernel can be attributed to the initialization of the various DOM structures needed by the SVG graphics engine when an application starts. Because of the way the DOM tree is represented, a significant amount of time is spent converting data from numbers to strings, and vice versa. None of this additional effort is required by the ScriptBrowser. Also note that the ScriptBrowser and the web browser in the Nokia N810 device use a different JavaScript virtual machine. Based on a number of micro-benchmarks, the web browser has a marginally better-performing JavaScript virtual machine than the plain-vanilla SpiderMonkey engine that we use in the ScriptBrowser. Nevertheless, the performance of typical (graphics intensive) Lively Kernel applications is dramatically better in the ScriptBrowser than in the browser-based mobile version of the Lively Kernel. This is largely due to the absence of the DOM and the native graphics API described in the next subsection.

## 5.2 Native graphics

In a conventional web browser, the JavaScript application communicates with a graphics engine primarily via the DOM, by reading and manipulating the various attributes (such as an object's color or fill attributes or coordinates) in the DOM tree. The graphics engine then responds to changes in the DOM tree by updating the display correspondingly. Attributes in the DOM tree are commonly represented as strings, which means that a considerable amount of time is spent converting various numeric parameters from numbers to strings and then (internally inside the graphics rendering engine) back to numbers again.

Another commonly used way for a JavaScript application to communicate with a rendering engine is to construct scripts or other executable structures (such as serialized objects) on the fly, and send them to the web browser for evaluation. For instance, a JavaScript engine could construct HTML strings on the fly, and send them to the web browser. Since this requires dynamic translation and interpretation of source code, this is considerably slower than performing conventional, direct graphics API calls.

Because the use of SVG (and indirectly the DOM) was considered one of the main culprits for performance issues of the Lively Kernel on a mobile device, we decided to rely on native graphics support instead of SVG. Most mobile devices today provide powerful graphics APIs, comparable to those APIs that were introduced in desktop computers in mid-1980s and 1990s. High-end mobile devices even include support for hardware accelerated graphics. Unfortunately, a web browser does not normally provide access to such powerful APIs.

In the ScriptBrowser, we replaced SVG with a JavaScript API that was mapped onto the underlying set of GTK and Hildon widgets available on the Nokia Maemo platform. Our

---

<sup>1</sup> Note that the ScriptBrowser and the web browser in the Nokia N810 device use a different JavaScript virtual machine. Therefore, performance results would not be directly comparable anyway.

JavaScript API provides a canvas-like drawing surface with low-level drawing primitives (similar to those offered by the Java™ ME MIDP Specification [RTV03]), augmented with a set of widgets – such as menus, pop-up windows, and device-specific notification windows – rendered natively. Support for a subset of Morphic widgets was implemented in JavaScript, so that Lively Kernel applications intended for the desktop version of the Lively Kernel can be run in the mobile version.

Allowing the scripts to utilize native widgets and a direct graphics API improved performance considerably. We have not been able to measure the performance differences precisely, but the overall performance improvement seems to be at least an order of one magnitude compared with the browser-based mobile version of the Lively Kernel.

The replacement of the SVG engine with a native graphics API was not without problems, though. One of the benefits of SVG is that it is a “managed” graphics system. In other words, an SVG graphics engine manages the entire state of the display itself, without requiring intervention or particular attention from the application developer. For instance, no explicit *repaint* calls or damage rectangle structures are needed, because the SVG engine is smart enough to keep track of the various layers of objects on display, and update the objects correctly when their attributes change. In contrast, in a conventional non-managed graphics API, it is the application's responsibility to keep track of the layers of objects on the screen, and repaint the objects explicitly when the objects change. If the repainting algorithm is not good enough, problems such as display flickering may occur.

In the early versions of the ScriptBrowser, a lot of flickering occurred when we were running applications that required a high display refresh rate, such as games. The problem was caused largely by our simplistic repainting strategy which did not keep track of damaged display areas (those that actually do require a repaint) in detail; as a result, essentially everything was redrawn when something on the display changed. Double buffering support was added to avoid the flickering problems. A simple damage rectangle algorithm was also implemented in JavaScript.

Another downside of abandoning SVG was that some of the more advanced features of the Morphic graphics architecture, such as object rotation, scaling, and gradient colors, could not be supported easily. While rotation and scaling (more generally: affine transformations) can be implemented with some relatively simple matrix calculations, and gradient colors can be supported using well-known filling algorithms, the use of native widgets (which on the Nokia Maemo platform do not support rotation, scaling or arbitrary colors) prevented various advanced Morphic features from being used. For instance, unlike in the desktop version of the Lively Kernel, the ScriptBrowser does not allow text, alert windows or menus to be rotated or scaled.

### *5.3 Event handling*

In the browser-based mobile implementation of the Lively Kernel, event handling is performed primarily by the web browser. Native events, such as `mousemove` or `keydown` events, are forwarded from the web browser to JavaScript applications in the form of DOM Event objects, and the Morphic framework (written in JavaScript) then generates synthetic JavaScript event objects that can be processed by the Lively Kernel application programmer.

The three layers required for each event (native event, web browser event object, and Lively Kernel event object) introduce a significant amount of overhead to event handling. While this approach works reasonably well on a desktop computer, it is not fast enough on a mobile device, especially when performing operations such as selecting and then dragging/moving a large number of objects simultaneously. In such situations, the mobile device and the relatively slow JavaScript virtual machine in the web browser simply do not have enough power to perform all the necessary event handling and repainting without generating a long backlog of unprocessed `mousemove` or other events.

In the ScriptBrowser, the event handling system has been streamlined to handle a larger number of events. In the absence of the DOM, the ScriptBrowser can convert native events arriving from GTK graphics directly into JavaScript event objects without dealing with intermediate DOM Event objects. The ScriptBrowser can also invoke JavaScript event handler callbacks directly without additional overhead. Unlike in the desktop version, the ScriptBrowser uses a singleton event object to minimize the number of generated objects and associated garbage collection. In a regular web browser, the event objects are effectively thrown away immediately after the events have been processed. The throwaway event objects place a significant burden on the garbage collector of the JavaScript virtual machine.

In order to avoid event backlog problems, an event filtering mechanism was added to the ScriptBrowser to analyze the incoming native event queue. For instance, when the number of unprocessed native `mousemove` events in the event queue exceeds a certain threshold, events are merged and only the latest one passed on to the JavaScript application. In the same fashion, duplicate `focus`, `blur` or `keydown` events or repaint requests are merged or combined to avoid congestion problems. This radically improves the observed performance, e.g., in situations in which the user drags a large number of objects or keeps a key pressed down for a long time.

Because we could not easily instrument or debug the code of the browser-based mobile version of the Lively Kernel, precise performance comparisons were not possible. However, based on informal benchmarks event handling performance is generally 2-3 times faster in the ScriptBrowser than in the web browser. Even with such an improvement, event handling performance is not yet always adequate. The remaining problems can be attributed mostly to the JavaScript virtual machine, which is not fast enough to process complex event handling and repainting sequences adequately.

## *5.4 Networking*

Asynchronous network communication is a central feature in all Web 2.0 technologies. In the absence of asynchronous networking, it would not be possible to support interactive, desktop-style applications without blocking the user interface while network requests are in progress. In web browsers, asynchronous HTTP networking is commonly provided by the XMLHttpRequest interface [XML08].

In the ScriptBrowser, we support asynchronous networking by implementing a subset of the XMLHttpRequest API that works over the wireless network connection offered by the Nokia Maemo platform. Since we did not aim for full web browser compatibility, we currently support

HTTP GET requests only, and allow data uploads to be performed using HTTP GET requests as well. Support for HTTP POST and PUT requests has not been completed yet. Improvements in the robustness of the networking operations are also needed, since it is still possible to occasionally crash the ScriptBrowser by performing complex network requests that download (or fail to download) a large amount of data.

Support for asynchronous networking in the ScriptBrowser was implemented using the underlying device-specific networking APIs and operating system processes. A new operating system process is spawned for each new asynchronous HTTP request, so that the JavaScript application can continue running while the network request is in progress. When a download is complete, the data is passed on to the main process running the actual JavaScript application. In principle, there are no restrictions on the number of parallel downloads that can be supported. However, the current implementation uses a static number of pending simultaneous requests, limiting the number of simultaneous requests to twenty requests. In regular web browsers, the number of simultaneously allowed network operations is considerably lower, usually only two.

Regarding the performance of networking operations, there are no noticeable performance differences between the ScriptBrowser and the web browser based mobile version of the Lively Kernel. Any possible differences are masked by networking delays that commonly dominate the performance of networking operations anyway.

### *5.5 Accessing other resources and interfaces*

In mobile devices there are various resources and interfaces that are not commonly available in desktop computers. Examples of such mobile-centric interfaces include text messaging, multimedia messaging, Bluetooth, camera, location, and telephony APIs. Unfortunately, no standardized, widely-used JavaScript APIs exist for these areas yet. Standardized JavaScript interfaces are also still missing for areas such as calendar support, address and/or phone book access, and various audio and media capabilities. It will likely take several years until JavaScript APIs will be available for these areas in the same way as such APIs are available for the Java™ Platform, Micro Edition (Java™ ME). Given how long standardization will usually take, it might be easier to simply adopt the Java ME APIs also for JavaScript. Those APIs are already widely established and adopted by all the major mobile device manufacturers.

In the ScriptBrowser, we have experimented with APIs for some of these areas based on our earlier work with the Java ME platform [RTV03]. To support offline applications and data, we have also made it possible to access the file system of the device using a straightforward POSIX-style file interface. In a conventional web browser this is impossible without additional libraries such as Google Gears (<http://gears.google.com/>). As will be explained in more detail in the next subsection, the security issues associated with such an API have been largely ignored so far.

### *5.6 Security*

The ScriptBrowser is an experimental research vehicle that was created primarily to gather experience about network-downloaded applications on mobile devices in the absence of various

constraints and performance issues that hamper the use of such applications in conventional web browsers. Because of such focus, security issues have been largely overlooked so far. In principle, the ScriptBrowser has a straightforward sandbox security model that allows access only to a limited number of APIs that are known to be safe. However, the presence of certain additional APIs such as the file API would make the system inherently vulnerable if it were used for real applications.

While security is a critical aspect of any system supporting network-downloaded applications and data, we feel that the current “one size fits all” security model of the web browser – built around the *Same Origin Policy* introduced originally in Netscape Navigator version 2.0 (<http://www.mozilla.org/projects/security/components/same-origin.html>) – is not ideal or sufficient for web applications [TMI08]. Rather, a more fine-grained security model granting different levels of access to different types of APIs and applications is needed. For instance, the current Same Origin Policy limitations make it unnecessarily difficult to create application mashups – web applications that combine code and data from various web sites into an integrated experience. Such applications have become increasingly popular in recent years.

In our work on web application security, we have been inspired by a number of proposals, including Google's Caja (<http://code.google.com/p/google-caja/>), Microsoft's MashupOS [WFH07] and IBM's SMash [KBS08]. A large number of security groups and communities – such as the Web Application Security Consortium (WASC), the Open Web Application Security Project (OWASP), and the W3C Web Security Context Working Group – are working on the problem. However, the creation of widely adopted, industry-wide standards is still likely to take years.

## 6. Discussion and Lessons Learned

During the work summarized in this paper, we have gathered a lot of experience in developing and using mobile web applications as well as in using the JavaScript language and the web browser as an application platform for mobile devices. In this section, we provide a summary of our experiences, starting from some general observations about the use and evolution of mobile devices for web applications, and then summarizing our observations about the web browser and JavaScript in mobile web application development.

**Evolution of mobile devices towards enabling mobile web applications.** The hardware used in mobile devices has rapidly become more powerful. When we originally started working on software applications for mobile devices in the mid-1990s, mobile devices had only a few tens of kilobytes of memory available for applications, and processor speeds barely reached ten megahertz. Today, it is not uncommon to find tens or hundreds of megabytes or even tens of gigabytes of memory in mobile devices. Processor clock speeds can be measured in the range of several hundred of megahertz. Network bandwidth for data access has rapidly increased from tens of kilobits in the late 1990s to tens of megabits per second today. At the same time, the devices have been opened up for third party software development. The Java ME platform, as well as the use of operating systems such as Linux or Windows Mobile have played a significant part in making software development for mobile devices increasingly similar to desktop software development.

We believe that over time, desktop and mobile application development will converge. In the long run, there will be content, services and applications that can be used from any type of terminal, including desktop computers and mobile devices. The problems arising from CPU speed differences, memory capacity differences, network bandwidth limitations, and different operating systems are likely to disappear. Eventually it will be possible to use the same web applications and services in desktop computers and other types of client devices.

However, the problems related to small screen sizes and usability will remain also in the future. Despite the increasing screen resolutions in mobile devices, there are still fundamental restrictions that impair the application porting efforts from desktop computers to mobile devices. These restrictions originate from the different physical characteristics and different types of input mechanisms in mobile devices, as well as from the fundamental differences in the way mobile devices are used compared with the relatively stationary use of desktop and laptop computers. With mobile devices, the user simply cannot be expected to establish long-lasting sessions with the applications in the same way as desktop applications are commonly used. Rather, most interactions with the mobile devices will be relatively short. In general, screen size limitations, different input mechanisms, and different device usage modalities will continue to have a significant impact on the use (and hopefully also the design) of applications.

**Using the web browser as a platform for mobile applications.** The web browser has rapidly become the most important means for accessing various types of information. The majority of the information that people need in their everyday lives is already readily accessible from a web browser. The importance of the web browser and the great popularity of mobile devices will dramatically increase the use of the web browsers on mobile devices. In the long run, the number of mobile web users will likely exceed the number of desktop computer users.

We have discussed our experiences in using the web browser as an application platform (in the desktop environment) in another paper [TMI08]. As summarized in that paper, there are problems in various areas such as usability, security, web browser compatibility, application deployment and performance. In the mobile space, problems are exacerbated especially in the areas of usability and performance. For now, mobile devices do not have enough horsepower to run serious applications inside an ordinary web browser. The regular web browser was not designed for high performance, and various structures and interfaces in it – especially those related to the DOM – make the browser unnecessarily slow as an application platform. Our own work on the ScriptBrowser shows that there is still a lot of room left for optimizing the browser, and we are confident that over time those problems will be resolved. For instance, a widely adopted Canvas API supporting graphics rendering directly inside the web browser would alleviate the problems considerably.

The usability problems related to mobile web usage have been investigated extensively elsewhere. For instance, the World Wide Web Consortium has published a useful best practices document [MVB08]. We will not repeat those findings here.

**Using JavaScript for mobile web applications.** Given that JavaScript support is included in every commercial web browser, the JavaScript language has rapidly become the *de facto* programming language of the Web. Although JavaScript was originally designed as a scripting language, i.e., a language targeted to simple scripting tasks, its use has rapidly spread into “real”

programming and serious applications consisting of tens of thousands of lines of code. We have summarized our experiences in using JavaScript as a real programming language in another paper [MiT07]. An excellent discussion on the good and bad aspects of the JavaScript language is provided in [Cro08].

In the mobile space the absence of widely established JavaScript APIs hinders the use of the JavaScript language for serious mobile application development. As we already mentioned earlier in Section 5.5, standardized JavaScript APIs are still missing for various mobile-specific areas such as text messaging, multimedia messaging, Bluetooth, camera, location, or telephony support. There are a large number of general-purpose JavaScript libraries available, including Dojo (<http://dojotoolkit.org/>) or Scriptaculous (<http://script.aculo.us/>), but those libraries do not address the specific needs of the mobile industry. As proposed in Section 5.5, it might be simplest if the JavaScript community adopted the Java ME libraries that have already been standardized by the mobile industry for most of the necessary areas.

The use of JavaScript for mobile web applications is hindered also by the poor performance of the current crop of JavaScript virtual machines. The purely interpretive nature of those virtual machines means that applications run several orders of magnitude slower than corresponding applications written in statically compiled programming languages such as C or C++. The dynamic memory management capabilities of the current JavaScript virtual machines are also arcane compared, e.g., to modern Java virtual machines. Long garbage collection pauses may still occur from time to time when running significant JavaScript applications. Thus, although the JavaScript language itself is fairly well suited for complex applications [MiT07], the underlying infrastructure components, in particular the virtual machines and libraries, are not. Fortunately, things will gradually improve with the introduction of more advanced, high-speed virtual machines such as Mozilla's Tamarin (<http://www.mozilla.org/projects/tamarin/>) and TraceMonkey (<https://wiki.mozilla.org/JavaScript:TraceMonkey>), Apple's SquirrelFish (<http://trac.webkit.org/wiki/SquirrelFish>) and Google V8 (<http://code.google.com/p/v8/>). There is a lot of existing expertise in building high-performance virtual machines, e.g., for the Java™ programming language; given the popularity of JavaScript, this expertise will eventually find its way also to the development of JavaScript virtual machines.

In general, our work on the ScriptBrowser has shown that there is a lot of relatively low-hanging fruit left in optimizing the environments for mobile web applications. By introducing a high-performance JavaScript virtual machine, and by enabling and standardizing more direct access to the necessary graphics and other interfaces inside the web browser, the performance of web applications could be improved dramatically. We think that it is quite possible to build an environment that would run web applications a few hundred times faster than the current web browsers, and simultaneously provide a user experience that is comparable to the best desktop applications.

## 7. Conclusion

In this paper we have summarized our experiences in porting the Sun Labs Lively Kernel web programming system from the desktop environment onto a Nokia N810 mobile device. The Lively Kernel is an exceptionally interactive, “zero-installation” web application development

platform that has been written entirely in JavaScript. It runs in an ordinary web browser without installation or browser plug-in components, supports desktop-style applications with rich user interface features and direct manipulation, and enables application development and deployment in a web browser using nothing more than existing web technologies.

The mobile porting of the Lively Kernel was completed to investigate the feasibility of running full-fledged web applications in mobile devices. Our work has been motivated by two major trends in the software industry today: the increasing use of the Web as an application platform, and the rapidly increasing availability of web-enabled mobile devices. Our belief is that in the long run mobile devices will become first-class execution environments for web services and applications. We also believe that the differences between the mobile and desktop web, apart from those related to usability, will eventually disappear. This will lead us towards “One Web” – with content, services and applications that can be used from any type of terminal, irrespective of the target environment.

We reported our porting experiences based on two different approaches that were used. First, we ported the Lively Kernel onto a regular web browser running in the N810 mobile device. Second, we developed a custom-built native execution environment called the ScriptBrowser that provides more direct and extensive access to the underlying resources of the mobile device. Based on our experiences, there are still a number of issues and challenges in running serious web applications on mobile devices. The majority of those issues arise from the poor performance of the web browser, and the fact that the web browser was not generally designed to be an application platform. However, apart from political and commercial issues, we do not see any fundamental technical reasons why those problems could not be solved. Our work on the ScriptBrowser shows that there is still a lot of room left for optimization.

In conclusion, we are excited by the rapid emergence of the World Wide Web as a platform for software applications. Web-based applications will open up entirely new possibilities for software development by making it possible to deploy applications instantly worldwide and to upgrade them easily. While the web browser is not an ideal platform for applications, the instant deployment aspect makes web applications inherently superior to conventional applications. With our own work on the Sun Labs Lively Kernel, we have demonstrated that there is a better way to build web applications that support rich user interaction, advanced graphics, and integrated development and application deployment. We hope that eventually those qualities will find their way also to the development of mobile web applications.

## References

- CPJ05 Crane, D., Pascarello, E, James, D., *Ajax in Action*. Manning Publications, 2005.
- Cro08 Crockford, D., *JavaScript: The Good Parts*. O'Reilly Media, 2008.
- Eic07 Eich, B. The Open Web – What's at stake. Browser Wars Retrospective: Past, Present and Future Battlefields, Panel Presentation at the South by Southwest Conference (SXSW2007, Austin, Texas, March 9-18), 2007.

- Fla06 Flanagan, D., *JavaScript: The Definitive Guide*, 5<sup>th</sup> Edition. O'Reilly Media, 2006.
- FPM08 Freedman, C., Peters, K., Modien, C., Lucyk, B., Manning, R., *Professional Adobe AIR: Application Development for the Adobe Integrated Runtime*. Wrox Publishing, 2008.
- Gri00 Griffith, A. *Gnome/Gtk+ Programming Bible*, IDG Books Worldwide, 2000.
- HTM08 *HTML 5 Specification*. World Wide Web (W3C) Consortium Working Draft, July 30, 2008, <http://www.w3.org/html/wg/html5/>.
- KBS08 De Keukelaere, F., Bhola, S., Steiner, M., Chari, S., Yoshihama, S., SMash: Secure Component Model for Cross-Domain Mashups on Unmodified Browsers. In *WWW'2008 Conference Proceedings, Web Client Security Track* (Beijing, China, April 21-25), 2008, pp. 535-544.
- IKM97 Ingalls, D., Kaehler, T., Maloney, J.H., Wallace, S., Kay, A., Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. Presented at the OOPSLA'97 Conference, Atlanta, Georgia, Oct. 5-9, 1997.
- IPU08 Ingalls, D., Palacz, K., Uhler, S., Taivalaari, A., Mikkonen, T., The Lively Kernel – A Self-Supporting System on a Web Page. In *Self-Supporting Systems Conference Proceedings* (Potsdam, Germany, May 15-16), *Lecture Notes in Computer Science LNCS 5146*, Springer-Verlag, 2008, pp. 31-50.
- Mal95 Maloney, J.H., *Morphic: The Self User Interface Framework*. Self 4.0 Release Documentation, Sun Microsystems Laboratories, 1995.
- MaS95 Maloney, J.H., Smith, R.B., Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of the 8th annual ACM Symposium on User Interface and Software Technology (UIST)*, Pittsburgh, Pennsylvania, 1995, pp. 21-28.
- MiT07 Mikkonen, T., Taivalaari, A., Using JavaScript as a Real Programming Language. Technical Report TR-2007-168, Sun Microsystems Laboratories, 2007.
- MVB08 Mobile Web Best Practices 1.0. World Wide Web Consortium (W3C) Recommendation Document, July 29, 2008, <http://www.w3.org/TR/mobile-bp/>.
- Mor08 Moroney, L. *Introducing Microsoft Silverlight 2.0*. Microsoft Press, 2008.
- RTV03 Riggs, R., Taivalaari, A., Van Peurseem, J., Huopaniemi, J., Patel, M., Uotila, A., *Programming Wireless Devices with the Java™ 2 Platform, Micro Edition* (2<sup>nd</sup> Edition). Addison-Wesley (Java Series), 2003.
- SVG03 Scalable Vector Graphics 1.1 Specification. World Wide Web Consortium (W3C) Recommendation Document, January 14, 2003, <http://www.w3.org/TR/SVG/>.
- TMI08 Taivalaari, A., Mikkonen, T. Ingalls, D., and Palacz, K., Web Browser as an Application Platform: The Lively Kernel Experience. Technical Report TR-2008-175, Sun Microsystems Laboratories, 2008.

- UnS87 Ungar, D., Smith, R.B., Self: the Power of Simplicity. In OOPSLA'87 Conference Proceedings (Orlando, Florida, October 4-8), ACM SIGPLAN Notices, Vol. 22, No. 12, 1987, pp. 227-241.
- VML98 Vector Markup Language. A proposal submitted to the World Wide Web Consortium (W3C) on May 13, 1998, <http://www.w3.org/TR/NOTE-VML>.
- WFH07 Wang, H.J., Fan, X., Howell, J., Jackson, C., Protection and Communication Abstractions for Web Browsers in MashupOS. In Proceedings of the 21st ACM Symposium on Operating Systems Principles (Stevenson, WA, USA, October 14-17), 2007, pp. 1-16.
- XML08 The XMLHttpRequest Object. World Wide Web Consortium W3C Working Draft, April 15, 2008, <http://www.w3.org/TR/XMLHttpRequest>.

## About the Authors

Dr. Antero Taivalsaari is a Principal Investigator and Senior Staff Engineer at Sun Labs. Antero is best known for his seminal role in the design of the Java™ Platform, Micro Edition (Java ME) – one of the most popular commercial software platforms in the world, with over two billion devices deployed so far. Antero has received Sun's Chairman's Award twice (in 2000 and 2003) for his work on Java ME technology. Since August 2006, Antero has been co-leading the Lively Kernel project with Dan Ingalls to bring desktop-style user experience to the world of web programming.

Prof. Tommi Mikkonen is a Visiting Professor at Sun Labs, and a professor of computer science at Tampere University of Technology, Finland. Tommi is a well-known expert in the area of software engineering. He has arranged numerous courses on software engineering and mobile computing. He recently published a book on mobile software development with Wiley & sons (Tommi Mikkonen, *Programming Mobile Devices: An Introduction for Practitioners*, John Wiley & sons, 2007).