

The Web Computer and Its Operating System: A New Approach for Creating Web Applications

Sergejs Kozlovičs^a

Institute of Mathematics and Computer Science, University of Latvia, Raina blvd. 29, LV-1459, Riga, Latvia

Keywords: Web Computer, Web Applications, Web Application Operating System, webAppOS, Web Application Platform.

Abstract: Web applications require not only more sophisticated infrastructure than traditional single-PC applications, but also a different way of thinking, where network-specific aspects have to be considered. In this paper, we introduce the web computer concept, which factors out network-related issues and provides an illusion of a single computer with directly attached CPUs, memory, and I/O devices. By assuming the web computer and its open operating system (webAppOS) as a target platform for web applications, developers can preserve the same level of thinking as when developing classical desktop applications. With this approach, which corresponds to the physiology of the human brain, web applications can be created faster. Besides, the proposed web computer specification can be viewed as a standardized environment (a Java Virtual Machine analog) for web applications.

1 INTRODUCTION

When computers were big, and programs were small, programmers lived according to the assumption that a single computer (C) will be operated by a single user (U) and will be used to execute one program (P) at a time. We call it the *three-singles assumption*. From the times of Ada Lovelace and Charles Babbage, this assumption continued to be true when the Turing/Post machine was defined in 1930-ties, and later, when the Harvard and von Neumann architectures appeared in 1940-ties. Today, with the broad availability of the Internet and the presence of modern hardware and operating systems capable of handling multiple users and being able to run thousands of concurrent tasks, the assumption is not true anymore. Nevertheless, when creating standalone desktop applications (i.e., having C as a premise), programmers still rely on the same thinking as in the three-singles assumption, since the OS factors out user management (U) and multitasking (P). That allows programmers to concentrate on their primary tasks, which has a positive impact on productivity, since the human brain is not capable of real multitasking, as recent neuroscience research reveals.


If we consider web applications, the single-computer part (C) of the assumption becomes unsat-

isfied, and the OS is not able to ensure it.¹ Therefore, developers of web applications have to adjust their thinking to consider multiple network nodes, where the resources (CPUs, memory, and I/O devices) are physically separated. Besides, different network nodes can have different environments (e.g., different CPU architectures, operating systems, or sets of attached devices), which adds to the complexity.

To facilitate the development of web applications, numerous excellent platforms have been developed, each having its own set of features and requirements. Such platforms usually factor out network-specific issues (such as user management, scalability, and security) and provide a convenient way to share data between network nodes. The webAppOS platform, proposed in this paper, is a step further – it brings the full three-singles assumption for web application developers by providing an illusion of a single computer. This approach has the following benefits:

- *Psychological.* It is easier for the human brain to think about one target computer instead of multiple network nodes.

¹The reason is simple: the OS is responsible only for directly attached resources (CPU, memory, and I/O devices). In both von Neumann and Harvard architectures, the network is represented by an I/O device (a network interface controller, NIC); thus, the scope of the OS is limited to one particular network node with an attached NIC.

^a <https://orcid.org/0000-0002-7085-383X>

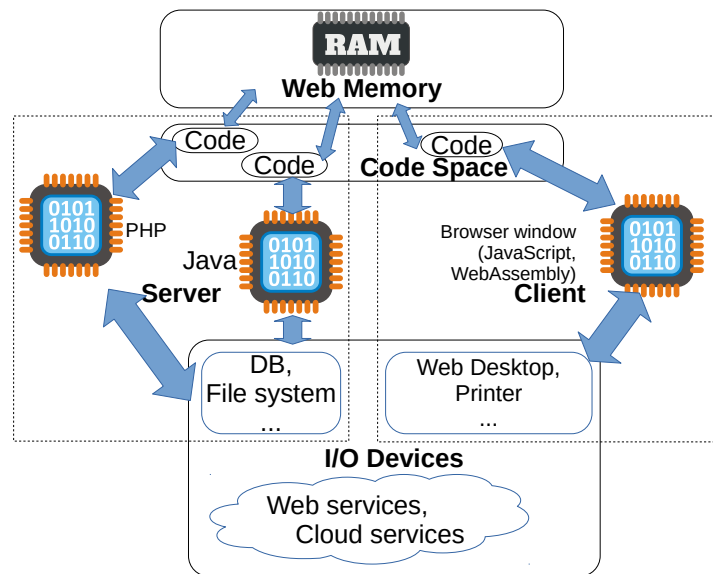


Figure 1: The main parts of the web computer architecture.

- *Technical.* webAppOS provides common grounds for web applications and services by factoring out technical issues in the same sense as Java Virtual Machine does so for classical applications.
- *Supervisory.* Being open-source, webAppOS facilitates the development of web applications that could be installed on a private server, thus, eliminating the risks of broken service URLs, sudden API changes, discontinued services, backdoors, and losing control on data (Stallman, 2010).

We start by defining the web computer concept, which brings the single computer illusion to the distributed web environment. In Section 3, we introduce webAppOS – the web computer “operating system”, which makes the single-computer illusion real. Section 4 mentions particular implementation details. Then the Related Work section compares the proposed architecture to existing approaches for developing web applications. Finally, we discuss some webAppOS benefits, outline further research directions, and conclude the paper.

2 THE WEB COMPUTER ARCHITECTURE

The **web computer** is an abstraction that factors out network-specific aspects and gives the developers of web applications the illusion of a single target computer with directly attached components. Like in classical computer architectures, the web computer has memory, processors, and I/O devices (see Figure 1).

2.1 Web Computer Memory

In the web computer, data memory differs from instruction memory (code memory), thus, resembling the Harvard architecture. The data memory is automatically synchronized between all involved network nodes (such as the client and the server), providing an illusion of a directly attached shared memory unit (RAM analog). Code is stored, executed, and sent via the network (when necessary) by means of existing classical technologies, e.g., via the `<script>` tag. We use the terms **web memory** and **the code space** for data memory and code memory, respectively.

Some of the arguments in favor of such Harvard-based design are:

- security considerations; we cannot trust the client in the web environment due to possible code injection attacks; since data are separated from the code, it is impossible for the client to inject code via the shared data memory (Andrews and Whitaker, 2006);
- different network nodes may have different execution environments; thus, code intended for one node can be meaningless for the other (e.g., the server can execute PHP, Java, and Python code, while the client runs JavaScript and WebAssembly code);
- web memory should not be wasted for code (data synchronization requires server-side RAM, which is a limited resource that has to be disseminated thrifty between all connected users);

- by relying on existing technologies for the code space, we do not need to “reinvent the wheel”.

Web Memory.

The main design choice regarding storing data in web memory is to represent data as a model (a graph-like structure), not as an array of bytes. A similar approach is used by Java Virtual Machine, where memory is represented by connected objects. The arguments in favor of such design choice are:

- Models and metamodels are easy formalized (e.g., by utilizing MOF or ECore), thus, making it possible to define a formal and platform-independent web memory specification (Steinberg et al., 2008; Object Management Group, 2011b).
- Models are close to OOP. Web memory objects can even be mapped the corresponding structures of the chosen OOP-style language in an ORM²-like way (Anuja, 2007; Ambler, 2019).
- Additional features such as memory consistency checks, reflection, and the granular undo/redo mechanism can be implemented on models (Kozlovics et al., 2011).
- The graph-like structure is more suitable for synchronization via the network: OOP-style memory objects can be created on different endpoints independently and then be merged, while the array-based web memory would require centralized management to avoid collisions during block allocation.

Because of synchronization overhead, web memory should be considered a limited resource, which is not intended to store large amounts of data.³ Web memory should be treated as a classical RAM analog for storing data currently in use, while larger data sets can be stored in web I/O devices (defined in Section 2.3).

The Code Space.

The web computer architecture permits usage of arbitrary technologies within the code space. However, to be able to invoke code in a uniform way, some formalization is needed. We assume that code is organized into actions. An **action** is a self-contained code fragment (e.g., a set of classes or modules with resolved dependencies), which implements certain useful functionality and has a specific **entry point**. Entry points are usually functions formatted according to certain conventions. Actions are identified by unique

²Object-relational mapping

³Compare that to Microsoft Office 365, which imposes the 5 MiB limit to files being edited online.

human-readable names, which map to the corresponding entry points. To avoid code injection attacks, all legitimate server-side actions must be registered, and the corresponding action code has to be validated that it will not harm the server. Only registered actions will be called at the server-side.⁴ We coin the term **web call** to denote a particular invocation of an action. We use the term **calling conventions** to indicate a way how the arguments are passed to an action, and how the value is returned. Currently, two calling conventions are supported: 1) passing the argument as an object via web memory (the return value, if any, is also stored somewhere in web memory) and 2) the argument and the return value are encoded as JSON objects (stringified in some cases).

To make a web call, the action name, the argument (according to the calling conventions) as well as other technical information (such as the connected user ID) has to be specified.⁵ All web calls are invoked in a standardized way, regardless of where the underlying action code is located. Thus, we say that web calls are **implementation-agnostic**, meaning that even if the implementation of an action is changed (e.g., moved from the server to the client and completely rewritten), all web calls to that action will remain valid as soon as the calling conventions and arguments match both the old and new implementations.

If the action code does not require an authenticated user session, the action is called **public** (private, otherwise). If the action code does not require access to web memory, the action is called **static** (non-static, otherwise). In the current specification, all non-static actions must also be private, since only authenticated users are allowed to access web memory.

Web memory can be compared to the global variable scope. It is accessible from all non-static actions in the given code space. All internal variables (regardless of their actual programming language-specific scope) used when implementing actions are considered local variables, — other actions are not able to access them.⁶

While web memory is not allowed to contain code, it can reference registered actions by their names; thus, these referenced actions can be eventually ex-

⁴We do not impose the same restriction on the client-side code since the user can get around this restriction easily.

⁵This technical information can be collected and filled automatically in most cases.

⁶To boost the execution of certain inter-connected actions, some environment-specific local cache can be introduced. Web calls of such coupled actions will be optimized to take advantage of the cache. However, the actions must also work correctly without the cache; thus, their common state should be stored by other means, e.g., in web memory or in some external shared database.

ecuted via web calls. Such name-based code references replace code pointers used in classical computer architectures.

2.2 Web Processors

Web-processors are server-side and client-side software units (not hardware units; therefore, there is no bijection with real processor cores) being able to launch code. We use the term **instruction set** to denote a set of hardware and software requirements that may be imposed by the code. Some examples are:

- a particular OS + processor architecture, e.g., “GNU/Linux-x64” or “Win32”;
- a particular OS with preinstalled services, regardless of the processor architecture, e.g., “Ubuntu 19.04” (both Intel and ARM targets satisfy this requirement);
- some higher-level technology (e.g., PHP, Java, Python, .NET, JavaScript, WebAssembly, etc.);
- presence of certain software or hardware (e.g., a printer).

By including/excluding version number and other requirements, different variations of instruction sets may appear, leading to the hierarchy of instruction sets. The hierarchy is based on the “subclass of” relation between instruction sets defined as follows:

an instruction set J is a **subclass of instruction set** I , iff code requiring environment I can be executed also within environment J .

Each web processor supports one or more instruction sets. By convention, a web processor implementing some particular instruction set should support also its superclasses (e.g., if “Client-Side JavaScript 6” is supported, “JavaScript 6” and “JavaScript” should also be supported).

There can be multiple web processors, each having its own list of supported instruction sets. In the classical client-server approach, the following web processors are normally present:

- at least one server-side web processor with several instruction sets supported out-of-the-box (e.g. for launching PHP, Java, and Python code); server-side web processors can switch between executing actions launched by different connected users;
- at the client side, there is usually one web processor being able to execute JavaScript and WebAssembly code within the browser window (other client-side technologies such as Flash, Java applets, and ActiveX can also be supported as separate instruction sets); each connected user has a dedicated client-side web processor.

Web processors running on additional servers (“remote web processors”) can also be added, when necessary (i.e., when some code requires different CPU architecture or operating system).

From the developer’s point of view, the web computer resembles a multiprocessor system, where web processors share the same data memory but have separate arithmetic and logic units (ALUs). However, the developer does not need to think about which web processor will be used to execute the given web call — that is managed seamlessly by webAppOS, resembling how the traditional OS manages multiple concurrent threads on a multiprocessor architecture.

2.3 Web I/O Devices

We use the input/output device metaphor to denote data sources and receivers other than web memory. We call them **web I/O devices**. Examples of such devices are the server-side file system and databases as well as external cloud storage. Specific graphical presentations within the browser window as well as client-side devices (such as printers) are also considered web I/O devices.

Web I/O devices can be accessed from the code via some web I/O device-specific API. While it is impossible to know and support in advance all possible web I/O devices, some of them are standardized in webAppOS by defining common APIs. One example is File System API, which not only provides access to user home directories located at the server side but also can be used to access external storage (e.g., cloud drives) mounted into the file system. Another example is Desktop API for accessing the web-based desktop analog inside the web browser. The API can be used to display predefined dialog windows or to show the list of installed web applications and launch them in new browser windows.

3 THE WEB COMPUTER OS

When dealing with real hardware resources, programmers rely on the OS API. Similarly, developers of web applications for the web computer rely on webAppOS, — an operating system analog, which provides access to web memory, web processors, and standardized web I/O devices via a set of predefined APIs.

3.1 webAppOS Web Applications

We define a webAppOS **web application** as a pair (UI, A) , where UI is a set of code artifacts that ensure

communication with the end user, and A is a set of registered actions that can be invoked via web calls.

Communication with the End User.

Like in classical applications, all possible user communication ways can be split into two groups: graphical and console. In graphical webAppOS applications, visual content is delivered via HTTP using some existing technique, e.g., as static HTML/CSS/JavaScript files or via server-side code generator (e.g., PHP). In console web applications, bi-directional web sockets are used for communication, where an additional graphical terminal-like web application has to be launched to display the corresponding input/output streams. To support all different ways of communication with the end user, webAppOS relies on **application adapters**. For instance, there can be an adapter serving plain HTML/CSS/JavaScript files, an adapter with the PHP interpreter, an adapter for console web applications, etc.

Actions.

Actions of a webAppOS application are implemented as classical code using existing technology stacks. The set of actions of a web application can contain server-side and client-side code as well as code requiring exotic instruction sets, which must be provided by remote web processors running on additional servers with specific environments installed. To support various platforms (e.g., desktop and mobile web browsers, or different processor architectures), multiple implementations of the same action can be created so that webAppOS can choose the implementation depending on the platform (thus, webAppOS benefits from the fact that web calls are implementation-agnostic).

Actions of a webAppOS application can:

- access web memory (non-static actions only);
- make web calls to invoke other actions (in an implementation-agnostic way);
- access standardized web I/O devices (such as the user's home directory within the file system, if the user has been authenticated);
- access other (non-standardized from the webAppOS point of view) devices in a platform-specific way;
- perform other computational tasks, etc.

Specific APIs, available for different environments, are provided by webAppOS to support the first three tasks above.

Some web applications do not need to access web memory. For instance, the login app, which displays

the login and password input fields, must be static, since unauthenticated users will not be allowed to access web memory. Such web applications can make web calls only to static actions.

3.2 webAppOS Projects

Let us consider a webAppOS application requiring web memory. Multiple users can be working with that application, and each user can access the application in different contexts (e.g., editing different documents). We use the term **project** to denote each such context. Projects in webAppOS resemble processes in traditional operating systems. However, there is a distinction: in projects, the code is normally idle and is executed only when web calls are made, but traditional OS processes are normally running but can sleep to wait for some signal. Like traditional operating systems isolate memory of different processes, webAppOS isolates web memory of different projects. We use the term **slot** to denote a web memory instance used by a particular project.

3.3 webAppOS Libraries

A webAppOS **library** is a set of registered actions that can be shared between different web applications. Data, however, are project-specific and stored within the corresponding project web memory slot. Thus, libraries are similar to dynamic libraries (DLLs) in traditional operating systems.

Libraries are useful for

- factoring out common actions;
- providing multiple implementations for graphical presentations or other platform-specific services that can vary depending on the platform. By factoring out platform-specific aspects into libraries, web applications can be developed in a platform-independent way, where particular implementations for library web calls are chosen at runtime.

3.4 webAppOS Services

A webAppOS **service** is a module that provides useful functionality to applications but is invisible to end users. In most cases, services do not provide the user interface⁷, but provide some web-based API that can be accessed programmatically (HTTP REST/AJAX or as non-HTTP service such as mail service). Since there can be different ways to launch services, webAppOS relies on **service adapters**. For

⁷Some technical user interface for developers and administrators can be provided, though.

instance, one adapter can launch services implemented as Java servlets, while some other adapter can launch Docker containers.

Services can make web calls and use webAppOS API to access web I/O devices or web memory slots of running projects, but some access validation has to be implemented for that. A service example is webAppOS webDAV service, which provides access to the file system via the webDAV protocol. The service relies on the webAppOS File System API and requires users to provide their credentials (login+password) to be able to access their home directories.

3.5 Scopes and Authentication

We use the term **scope** to denote a resource (or a set of resources) that can be accessed only by authenticated users. Each scope has a name defined by the resource provider. For instance, Google defines multiple scopes such as “profile”⁸ and “spreadsheets”⁹; webAppOS also defines certain scopes, e.g., “login”, which can be used to access user’s home file system and make private web calls.

Before accessing a resource, the scope must be authenticated. A universal API (the Scopes API) is provided by webAppOS for that. The Scopes API relies on vendor-specific **scopes drivers** (such as the “google_scopes” driver), which perform authentication and receive access tokens (keys). Since authorizing scopes requires user’s intervention at the client-side, the Scopes API is available only at the client-side. Once the token is obtained, it is usually stored in some known place at the client side, at the server side, or at both sides (depending on the driver), and can be used by webAppOS applications, services, and web I/O devices to access the resource.

3.6 Drivers

Scopes drivers can contain also drivers for certain I/O devices (**web I/O device drivers**). For example, the Google scopes driver can be shipped with a web I/O device driver implementing the webAppOS File System API for Google Drive. Some drivers are bundled with webAppOS, others are provided by third-parties.

Drivers for web I/O devices implementing standardized APIs can be used by webAppOS to extend its capabilities. For instance, by using file system drivers, webAppOS can mount remote file systems as subdirectories in the webAppOS file system. Thus,

⁸<https://www.googleapis.com/auth/userinfo.profile>

⁹<https://www.googleapis.com/auth/spreadsheets>

whenever webAppOS File System API is called (directly or via the webDAV service), the mounted directory will be accessible as any other native directory.

3.7 webAppOS Execution Environments

Typically, webAppOS runs as a web server software, and users access it from client browsers. We say then that webAppOS runs in the **web** environment. By bundling the web server and a web browser component into a single desktop application, webAppOS web applications can be launched as standalone desktop applications (**desktop** environment). If graphical presentations are re-designed (re-written) to fit into a smaller screen with support for touch events, we can talk about the **mobile** environment.

As a special use case, webAppOS can be used to develop lightweight applications that do not require any webAppOS server-side code, but can rely on client-side API and client-side drivers to access third-party cloud services. We call such applications **serverless**. They must meet the following criteria:

- they have to be deployed as plain HTML/CSS/JavaScript files, which can be served by the simplest web server or launched from a local directory;
- they may not rely on webAppOS server-side API, but can use client-side API and client-side web calls;
- to access third-party scopes, they must rely on client-side tokens only.

We can think of webAppOS execution environments as of different targets for web applications, where all targets can share the same code (perhaps, with slight variations that can be factored out by webAppOS). Thus, webAppOS can be compared to Java Virtual Machine that can execute the same Java bytecode in different environments (with some platform-specific variations factored out by the VM).

4 IMPLEMENTATION

Figure 2 depicts the implementation of webAppOS, split into components (details are provided further in this section). All components are original, but some of them rely on third-party libraries (e.g., drivers for accessing third-party resources). The server-side part uses Java as the primary language, while the client-side part uses JavaScript. Using Java at the server side is reasonable since Java does not suffer from

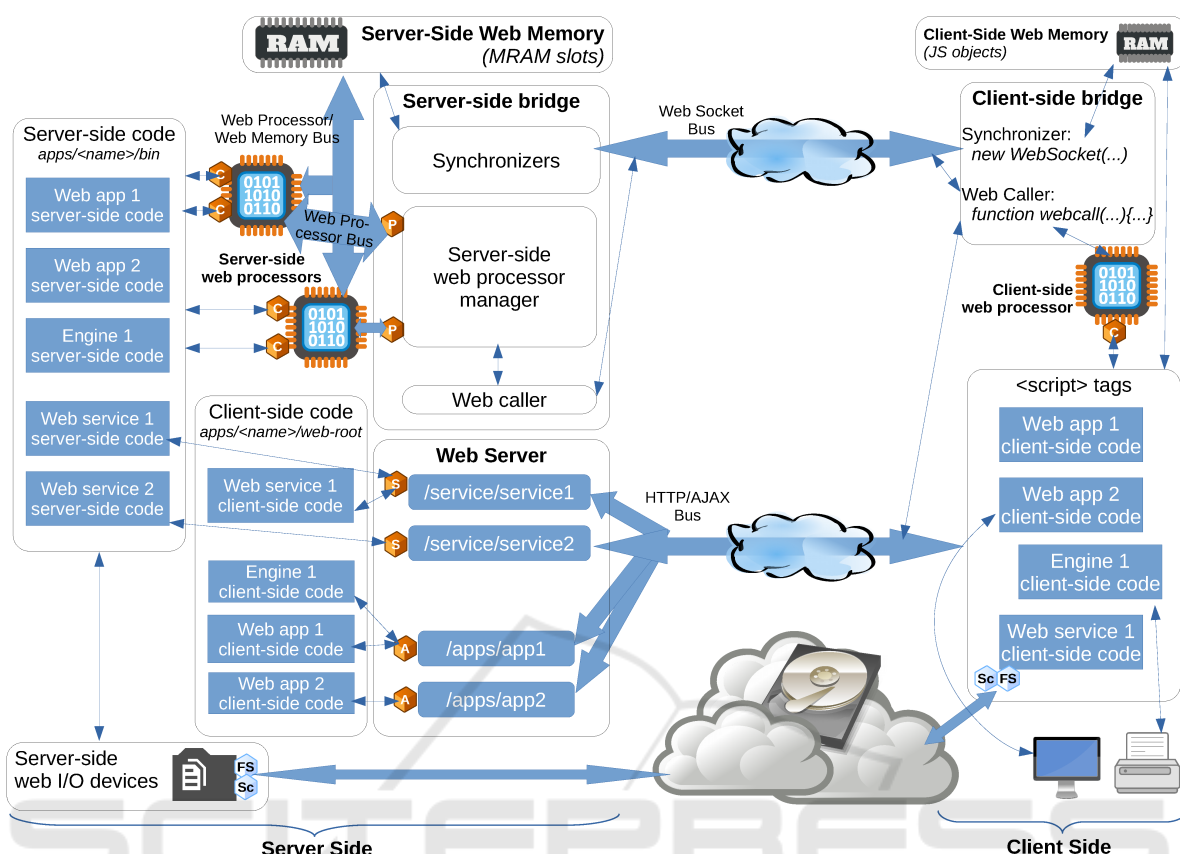


Figure 2: Implementation of the web computer architecture. Buses are depicted by thick arrows. Small cubes denote drivers and adapters (“P” and “C” for web processor and web call adapters, “S” and “A” for service and application adapters, “Sc” and “FS” for scopes and file system drivers).

popular security breaches based on stack overflow or char* overflow. Server-side drivers and adapters also have to be implemented in Java, but they can invoke non-Java code via JNI or inter-process communication. Similarly, client-side drivers and adapters must be implemented in JavaScript, but they can call non-JavaScript code (e.g., WebAssembly or Flash).

4.1 Web Memory

Web memory has been implemented via the model repository AR, which uses an efficient encoding of models (Kozlovičs, 2018; Kozlovičs, 2019). The models are stored in memory-mapped files, thus, making it easier to share web memory with server-side web processors. The encoding of models is suitable for direct synchronization via web sockets. AR can be accessed via universal Repository Access API, RAAP¹⁰, from Java as well as from native code (via a cross-platform dynamic library).

¹⁰<http://webappos.org/dev/raapi/>

4.2 Web Processors

Server-side and remote web processors are launched as separate OS processes via corresponding **web processor adapters**. The idea of launching web processors as separate OS processes has the following important benefit: if the code that is being executed within a web processor crashes, the process can be re-launched without affecting other web processors and web server software. Furthermore, if code freezes (or takes too long to execute), webAppOS can forcefully terminate the web processor and then relaunch it for executing other code. However, upon web processor termination, the corresponding web memory slot has to be invalidated (this resembles an ordinary application crash when data from RAM are being lost).

Each user runs the default client-side web processor implemented in JavaScript. Both server-side and client-side web processors rely on **web call adapters**, which correspond to available instruction sets. Each web call adapter is able to execute code designed for the supported instruction set and to pass arguments to

the action entry point according to some calling conventions. When a web call is made, the following information is analyzed and passed to the appropriate web processor and then to the appropriate web call adapter:

- the action name;
- the action declaration consisting of the instruction set, code location, calling conventions as well as other technical information (such as whether the web call is static and/or private);
- the passed argument(s) according to the calling conventions of that action.

4.3 The Code Space

The code space is represented by the *apps* directory, located or mounted at the server side. Each subdirectory corresponds to some webAppOS application or service containing both server-side and client-side code. Server-side code (if any) is usually located in the *bin* sub-subdirectory. Traditional technologies are used for such code (e.g., Java code is stored in .class or .jar files). Client-side code (if any) is usually stored in the *web-root* sub-subdirectory, which is automatically mapped to the corresponding web application or web service URL (e.g., “/apps/myapp”) accessible from the web browser. To map applications and services to URLs, webAppOS relies on application and service adapters.

To support load balancing, the same code space can be shared between multiple webAppOS instances using classical technologies, e.g., via a network file system or by directory replication. A similar approach can be used to share webAppOS file system containing users’ home directories.

4.4 Web I/O Devices

For predefined web I/O devices webAppOS provides Java and JavaScript APIs. The same APIs can be used to access third-party devices of the same type. For example, webAppOS File System API can be used to access various cloud drives in the same way as the home file system. Remote file systems can even be mounted within the home file system.

However, before accessing any third-party web I/O device (predefined or exotic), the scopes driver has to be loaded and the corresponding scope requested via the Scopes API. After the scope has been authenticated and access tokens obtained, webAppOS and applications can use remote resources via corresponding web I/O device APIs (predefined or exotic), which should be implemented in web I/O device drivers bundled with the scopes driver.

Scopes drivers are implemented as webAppOS services, which usually contain both client-side code (e.g., for displaying the third-party authentication window) and server-side code (e.g., for validating and storing access tokens) as well as drivers code. For serverfull mode, only server-side web I/O device drivers are required. To support the serverless mode as well, the scopes driver should provide also an independent client-side implementation of device drivers.

4.5 Bridges

The client and the server, each has a module called a **bridge**, which is responsible for synchronizing web memory, managing web processors at the given network node¹¹, and processing web calls. When one side makes a web call that has to be executed at the other side, the bridges communicate on how and where to execute code. The server-side and client-side bridges can be compared to the north and south bridges in the traditional motherboard layout, where different devices can be attached to different bridges, but there is a communication channel between the bridges.

4.6 Buses

Besides the main components, there are also several communication channels for transferring data between them. While some channels are implemented as simple function calls, others (which we call **buses**) require sending data between different OS processes or between different network nodes. There are the following main buses in webAppOS:

- *HTTP/AJAX Bus* is used to deliver client-side code and user interface (HTML/CSS/JavaScript) from the server to the client browser; provides access to webAppOS web services or their administrative settings;
- *Web Socket Bus* ensures web memory synchronization and forwards web calls when they have to be executed on the other node; this is the main communication channel between webAppOS bridges;
- *Web Processor/Web Memory Bus* is used to access web memory slots from server-side web processors (which are separate OS processes); implemented via memory-mapped files;
- *Web Processor Bus* is a communication channel between the bridge and server-side and remote

¹¹Currently, the client-side bridge has only one web processor; thus, a web processor manager is not needed there.

web processors; implemented via Java RMI (Pitt and McNiff, 2001).

5 RELATED WORK

The idea of providing an operating system-like environment for the web is not new (Lawton, 2008). Web desktops are desktop-like environments in a web browser window. Sometimes they are called web OSes; however, they resemble window managers, not full operating systems. As of 2019, many have been discontinued (e.g., eyeOS, ZeroPC). Others such as Os.js¹², WebDesktop.biz¹³, and AaronOS¹⁴ are alive, but not widely used. The proposed webAppOS comes with a preinstalled Desktop application, which acts as a window manager. However, alternative launchers of web applications are also possible (e.g., via a menu button showing the icons of the installed applications, like implemented in Microsoft Office365 or Google Docs).

While web desktops are not popular today, developers focus on creating rich HTML-based pages and single-page web applications. Since HTML was initially designed for static documents, numerous frameworks appeared, which add dynamics to HTML. The main goal is to support single-page applications at the client side (running in the browser) via a set of useful libraries, templating mechanism, and certain design patterns. Examples include AngularJS¹⁵ (and its successor Angular2+¹⁶), Dojo Toolkit¹⁷ (and its successor Dojo2¹⁸), React¹⁹ (for building UI), Aurelia²⁰, Ember²¹, Vue²², and Backbone.js²³. Classical libraries such as jQuery²⁴ and jQueryUI²⁵, Bootstrap²⁶, and D3²⁷ can also be mentioned. However, from webAppOS point of view, all these frameworks are just client-side libraries, which can be used to implement actions at the client side. Although client-

side technologies mentioned above are mainly targeting JavaScript, some support TypeScript. Java programs can also be cross-compiled to JavaScript by Google Web Toolkit, GWT²⁸. The Blazor²⁹ toolkit allows developers to write web applications in C#, which can be then compiled to WebAssembly, which is now supported by most modern browsers.

There are numerous web application frameworks (including ASP.NET³⁰, JavaServer Faces³¹, etc.) that are good for business-oriented form-based web applications. However, each such framework is tied to particular specific technologies, and certain framework-specific knowledge is required to create web applications based on the chosen framework. In contrast, our web computer architecture does not impose environment-specific restrictions on web applications. Developers can continue to use familiar technologies. The only exception is webAppOS itself, where drivers and other system modules have to implement certain Java or JavaScript APIs (but their internal implementation is not restricted).

Node.js³² provides JavaScript environment outside the web browser; thus, JavaScript code can be launched at the server side, providing the same language environment for both the client and the server. Unlike Node.js, webAppOS does not enforce JavaScript, neither impose the asynchronous execution to applications; it relies on web processors instead, where computation-heavy actions (which can be synchronous) can be executed in parallel using virtually any programming language. However, Node.js has many benefits such as an excellent community, many useful libraries, and a package manager. Thus, Node.js can be launched as a web processor within webAppOS.

Meteor³³ is a Node.js-based framework for the rapid creation of web applications in JavaScript with automatic client-server synchronization. Perhaps, it is the closest webAppOS sibling. However, Meteor uses a NoSQL database MongoDB, which is optimized for fast queries, but not for fast writes, thus, making a bottleneck, when used to simulate a read-write RAM analog. Furthermore, Meteor requires explicit data listeners and is tied to JavaScript. Regarding scalability, Meteor is tailored to the commercial Galaxy cloud. The proposed webAppOS is an open-source project aiming to support different clouds and various existing web services.

¹²<https://www.os-js.org/>

¹³<http://webdesktop.biz/>

¹⁴<https://aaron-os-mineandcraft12.c9.io/aosBeta.php>

¹⁵<https://angularjs.org/>

¹⁶<https://angular.io/>

¹⁷<https://dojotoolkit.org/>

¹⁸<https://dojo.io/>

¹⁹<https://reactjs.org/>

²⁰<https://aurelia.io/>

²¹<https://emberjs.com/>

²²<https://vuejs.org/>

²³<https://backbonejs.org/>

²⁴<https://jquery.com/>

²⁵<https://jqueryui.com/>

²⁶<https://getbootstrap.com/>

²⁷<https://d3js.org/>

²⁸<http://www.gwtproject.org/>

²⁹<https://blazor.net/>

³⁰<https://dotnet.microsoft.com/apps/aspnet>

³¹<http://www.java-serverfaces.org/>

³²<https://nodejs.org/>

³³<https://www.meteor.com/>

Google Apps Script³⁴ is a platform for developing web applications intended to be run in a web browser. Google Apps Script integrates with Google services. However, if we need certain server-side computation or database, we need to create a web service or some API for that.

Google Chrome OS³⁵ pursues a different goal – to make a browser act as an operating system, thus, making the software dependent on a particular browser. In contrast, *webAppOS* will support different web browsers, different server-side operating systems as well as different cloud services.

CloudRail Unified APIs³⁶ was a commercial initiative (with a free community version) for providing universal APIs for cloud services such as storage, messaging, payments, etc. Unfortunately, support for CloudRail APIs was discontinued on March 1, 2019. Since CloudRail APIs align to *webAppOS* web I/O device APIs, *webAppOS* drivers implementing similar unified APIs can eventually be developed.

CORBA is a set of standards proposed by OMG for enabling interoperability between diverse programming languages and operating systems (Object Management Group, 2011a). Originated in 1991, it was an excellent idea to implement web services via distributed objects. However, CORBA implementations proved to be heavyweight and slow. Although still available, it is considered deprecated (CORBA has been even removed from Java11). Thus, to support inter-process communication, *webAppOS* relies on a more lightweight Java RMI protocol for making calls between the server-side bridge and web processors (Pitt and McNiff, 2001). To synchronize web memory, *webAppOS* relies on web sockets, which use lower resources than CORBA or traditional REST/AJAX requests.

An interesting approach for bringing traditional desktop applications to the web is via cloud platforms such as RollApp and AlwaysOnPC³⁷. Each connected user runs their own copy of some classical application at the server side, but application windows are being forwarded to the web browser implementing the X server (or its alternative). A similar approach is used in open-source Gnome Broadway³⁸ and xpra³⁹. Commercial Citrix Virtual Apps⁴⁰ (for-

merly XenApp and XenDesktop) provide access to Windows, Linux, and SaaS applications in a way similar to Microsoft Remote Desktop. Technically, the approach of forwarding a remote window can be implemented in *webAppOS* by assigning to each connected user a server-side web processor running the given desktop application. Web memory can be used to exchange information on how to connect to the X server running within a web browser. Input and output streams of console applications can be forwarded to the web browser in a similar way. However, with such a forwarding-based approach, each web processor will be reserved for the whole user session, not just one web call.

Since web memory is represented by a model, which, in essence, is a graph, it can be easily mapped to linked data and semantic web technologies such as RDF and OWL (W3C, 2014b; W3C, 2014a; W3C, 2004; W3C, 2012). We can even introduce specific web processors implementing semantic reasoners, which can be considered specific instruction sets (Corno and Farinetti, 2012).

Electron⁴¹ is a convenient framework for developing cross-platform desktop applications by utilizing web technologies. Electron comes with a bundled web browser and a set of APIs for accessing native OS functionality. Electron resembles how *webAppOS* is intended to support multiple target environments (web, desktop, and mobile). However, *webAppOS* does not restrict the developers to use JavaScript, – virtually any technology available for the target platform can be used.

Although some approaches to bring the classical single-PC thinking to the process of developing web applications have been published, they required program re-structuring (Matthews et al., 2004). In *webAppOS*, existing technologies and even existing code can be re-used. The only requirement is that the code has to follow the calling conventions of the corresponding web call adapter.

To summarize, while there are excellent technologies that can solve particular tasks occurring during the development of web applications, no project is aimed to provide a centralized infrastructure to tie all these technologies into one ecosystem. Currently, this resembles how GNU had excellent software but lacked the OS kernel. Initially, GNU aimed to adopt the Hurd kernel, but Linux proved to be more viable (Tozzi, 2017). Perhaps, *webAppOS* can eventually become “a kernel” (or a Java Virtual Machine analog) for web technologies. However, we have to admit that to earn the market, significant efforts from the open-

³⁴<https://www.google.com/script/start/>

³⁵<https://www.google.com/chromebook/>

³⁶<https://cloutrail.com/>

³⁷<https://www.rollapp.com/>,
<http://www.alwaysonpc.com/>

³⁸<https://developer.gnome.org/gtk3/stable/gtk-broadway.html>

³⁹<https://xpra.org>

⁴⁰<https://www.citrix.com/products/>

[citrix-virtual-apps-and-desktops/](https://www.citrix.com/products/)

⁴¹<https://electronjs.org/>

source community as well as collaboration with cloud service companies are required.

6 BENEFITS

One of the main benefits of webAppOS is the presence of automatically and transparently synchronized web memory. Web memory acts as a shared data model between the client and the server. The presence of such a model is essential for single-page applications, SPAa, where the DOM has to be updated when the model changes (Takada, 2013). SPAs built upon webAppOS can also access standardized web I/O devices (either bundled within webAppOS or third-party) in a uniform way. Serverless SPAs can also benefit from using webAppOS: they can use webAppOS serverless drivers for accessing third-party resources via unified client-side webAppOS APIs.

Another significant webAppOS benefit is the ability to invoke code via web calls in an implementation-agnostic way. The developers do not need to be aware of where the code is located and how it will be executed: the process is managed by webAppOS and its bridges. Thus, the code implementing a particular web call can be moved or re-written, without altering the main web application. The performance slowdown of code running within webAppOS is minimal. The slowdown occurs only when the code accesses web memory due to triggered synchronization. However, the overhead is minimal since data are synchronized without re-encoding, and the synchronizer is launched in a separate thread, thus, allowing the main thread to continue independently on the synchronization process.⁴² This differs from CORBA and similar technologies, where communication between network nodes requires serialization, deserialization, and (usually) the full roundtrip (Object Management Group, 2011a).

Alan Kay, a Computer Science pioneer, once said that the web browser is, in essence, a mini-operating system. We would say that webAppOS is a superstructure over *both*, the server-side OS and the client-side web browser (the “mini-OS”). Such a superstructure makes the web computer illusion possible. Physical locations of the code and memory are hidden beyond webAppOS APIs. With webAppOS, developers can continue to use familiar technologies for writing code, but they do not have to think about the network.

⁴²The only noticeable delay can be observed between subsequent web calls executed in different network nodes since web memory changes have to be physically propagated from the first to the second node.

This is a strong psychological advantage since developers can stay within the three-singles assumption and create web applications at the same speed as when creating desktop applications. Existing standalone applications can also be easily converted to web applications by following these simple steps: defining a set of actions for invoking them via web calls, re-designing GUI using browser technologies, and adjusting webAppOS configuration.

7 CONCLUSION

The paper provided insight into webAppOS, a platform that ensures the three-singles assumption for developers of web applications and services. The proposed platform factors out almost all web-specific aspects; therefore, developers can assume they are writing applications for a single PC, with single memory and multiple processors attached directly. Network communication, user authentication, automatic acquisition of SSL certificates and their renewal via the ACME protocol, access to user’s home directory at the remote file system (where files can be downloaded from/uploaded to), and other features, are provided by *webAppOS* automatically, thus, simplifying the system administrator’s work. From the end-user experience, webAppOS resembles Google Drive, Microsoft OneDrive, and Apple’s iCloud, but in webAppOS, third-party web applications can be installed as well.

The webAppOS sources and the demo are available at webappos.org. The full specification is available at webappos.org/theory, and the API documentation can be found at webappos.org/doc.

We are working on migrating our desktop-based ontology editor OWLGrEd to the web environment by using webAppOS (<http://owlgred.lumii.lv>, (Barzdins et al., 2010)). We are also planning to migrate the data analysis tool Data Galaxies to webAppOS. These tools are intended to prove the feasibility of the approach.

Currently, webAppOS has been implemented for the web platform. We are planning to release the webAppOS distribution for the desktop environment soon, where the “write once, run everywhere” principle will be respected.

Although the current webAppOS specification has been developed with scalability in mind, a standardized set of APIs has to be published for that, certain design choices have to be made, and support for existing cloud services (such as Amazon EC2, Google Cloud, and OpenStack) has to be provided. In addition, we have to pay more attention to privacy issues and low requirements. These are topics for further re-

search.

ACKNOWLEDGMENTS

The work has been supported by European Regional Development Fund within the project #1.1.1.2/16/I/001, application #1.1.1.2/VIAA/1/16/214 “Model-Based Web Application Infrastructure with Cloud Technology Support”.

REFERENCES

- Ambler, S. (2002–2019). Mapping objects to relational databases: O/R mapping in detail. <http://www.agiledata.org/essays/mappingObjects.html>.
- Andrews, M. and Whittaker, J. A. (2006). *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services*. Addison-Wesley Professional.
- Anuja, K. (2007). *Object Relational Mapping*. PhD thesis, Cochin University of Science and Technology.
- Barzdins, J., Barzdins, G., Cerans, K., Liepins, R., and Sprogis, A. (2010). OWLGrEd: a UML style graphical notation and editor for OWL 2. In *Proceedings of OWLED 2010*.
- Corno, F. and Farinetti, L. (2012). Logic and reasoning in the semantic web (part II – OWL). Materials for the “1LHVIU - Semantic Web: Technologies, Tools, Applications” course at *Politecnico di Torino, Dipartimento di Automatica e Informatica*. <http://elite.polito.it/files/courses/01LHV/2012/7-OWLreasoning.pdf>.
- Kozlovics, S., Rencis, E., Rikacovs, S., and Cerans, K. (2011). A kernel-level UNDO/REDO mechanism for the Transformation-Driven Architecture. In *Proceedings of the 2011 conference on Databases and Information Systems VI: Selected Papers from the Ninth International Baltic Conference, DB&IS 2010*, volume 224 of *Frontiers in Artificial Intelligence and Applications*, pages 80–93, Amsterdam, The Netherlands. IOS Press.
- Kozlovičs, S. (2018). Efficient model repository for web applications. In *Proceedings of the 13th International Baltic Conference on Databases and Information Systems (Baltic DB&IS 2018)*, volume 838 of *CCIS*. Springer Nature Switzerland.
- Kozlovičs, S. (2019). Fast model repository as memory for web applications. *Databases and Information Systems X*, 315:176–191.
- Lawton, G. (2008). Moving the os to the web. *Computer*, 41(3):16–19.
- Matthews, J., Findler, R. B., Graunke, P., Krishnamurthi, S., and Felleisen, M. (2004). Automatically restructuring programs for the web. *Automated Software Engineering*, 11(4):337–364.
- Object Management Group (1991–2011a). Common Object Request Broker Architecture. <http://www.corba.org/>.
- Object Management Group (2011b). OMG Meta Object Facility (MOF) Core Specification Version 2.4.1.
- Pitt, E. and McNiff, K. (2001). *Java.Rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Stallman, R. (2010). Who does that server really serve? <http://www.bostonreview.net/richard-stallman-free-software-DRM>.
- Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008). *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley.
- Takada, M. (2013). Single page apps in depth. <http://singlepageappbook.com/>.
- Tozzi, C. (2017). *For Fun and Profit: A History of the Free and Open Source Software Revolution*. The MIT Press.
- W3C (2004). OWL Web Ontology Language reference. <http://www.w3.org/TR/owl-ref/>.
- W3C (2012). OWL 2 Web Ontology Language document overview (second edition). <http://www.w3.org/TR/owl2-overview/>.
- W3C (2014a). RDF Vocabulary Description Language 1.0: RDF Schema. <http://www.w3.org/TR/rdf-schema/>.
- W3C (2014b). Resource Description Framework. <http://www.w3.org/RDF/>.