webAppOS: Creating the Illusion of a Single Computer for Web Application Developers

Sergejs Kozlovičs(✉)

Institute of Mathematics and Computer Science, University of Latvia,
Raina blvd. 29, Riga 1459, Latvia
sergejs.kozlovics@lumii.lv

Abstract. Unlike traditional single-PC applications, which have access to directly attached computational resources (CPUs, memory, and I/O devices), web applications have to deal with the resources scattered across the network. Besides, web applications are intended to be accessed by multiple users simultaneously. That not only requires a more sophisticated infrastructure but also brings new challenges to web application developers.

The webAppOS platform is an operating system analog for web applications. It factors out the network and provides the illusion of a single computer, the "web computer". That illusion allows web application developers to focus on business logic and create web applications faster. Besides, webAppOS standardizes many aspects of web applications and has the potential to become a universal environment for them.

Keywords: Web computer · Web applications · Web application operating system · webAppOS · Web application platform

1 Introduction

Babbage and Turing assumed that the computer is as a single device executing one program at a time and operated by a single user. Such a way of thinking is close to the psychology of the human brain since the brain is not able to focus on multiple tasks at the same time. Today, however, multitasking, networking, and multiple concurrent users are common as air. Luckily, modern operating systems implement multitasking, multiuser management, and local resource and device management. This aids in creating single-PC desktop applications, but does not help with web-based applications since the developers still have to think about application-level protocols as well as how to manage resources (CPUs, memory, and I/O devices) scattered across the network. The question arises: is it possible to simplify the process of developing web applications by allowing the developers to retain the Babbage/Turing way of thinking?

In our recent publication, we defined the concept of the *web computer*, the illusion of a single logical computer for web applications [9]. Although it still

© Springer Nature Switzerland AG 2020
A. Bozzon et al. (Eds.): WEBIST 2019, LNBIP 399, pp. 1–21, 2020.
https://doi.org/10.1007/978-3-030-61750-9_1

requires multiple physical network nodes to operate, web applications do not access them directly, but via the intermediate layer, webAppOS (web application operating system), which makes the illusion possible.

We recall the web computer architecture and the main functions of webAppOS in the next two sections. In this paper, we extend our contribution by providing additional details on webAppOS implementation. We also describe the process of creating a new web application from scratch as well as the process of migrating two real applications, OWLGrEd and DataGalaxies, to webAppOS. Furthermore, in this paper, we also address scalability issues. We conclude by discussing how webAppOS differs from existing Google Docs-like platforms.

2 The Web Computer

The **web computer** is an abstraction that hides network communication and creates the illusion of a single computer. The web computer consists of the following main parts: web memory (data memory), the code space (instructions memory), web processors, and web I/O devices.

Notice that data and code memory are separate; thus, the web computer follows the Harvard architecture. That differs from most classical computers, which follow the von Neumann architecture, where data and instructions are put into the same memory. The main reason for the Harvard-based approach is security: web applications are subject to code injection attacks [1]. We intentionally protect server-side code from being altered via web memory by untrusted clients. Nevertheless, approved references to code (code pointers, which actually are strings) can be stored as data in web memory. Another reason for the Harvard-based approach emerges from the differences between existing server-side and client-side environments (e.g., server-side PHP code is meaningless for the web browser). Thus, while we can synchronize the data transparently to create the illusion of a single data memory unit, it is not necessary to synchronize code.

We continue by describing the main parts of the web computer.

2.1 Web Memory (Data Memory)

Web memory is represented by a formal model. It consists of classes and objects (class instances). Classes have attributes, while objects have attribute values. Besides, there are associations between classes and the corresponding links between objects. Thus, web memory is an OOP-like structure, similar to that used by Java Virtual Machine.

The main reason for such design choice comes from the fact that models, in essence, are graphs; thus, they are more suitable for synchronization than classical arrays of bytes. Besides, models can be easily formalized (e.g., using standards like MOF and ECore [11,16]).

Since synchronization involves some overhead, web memory should not be wasted. It is an analog of classical RAM; thus, only data that are currently in use should be stored there. Larger data sets can be stored elsewhere, e.g., in web I/O devices (see Sect. 2.4).

Multiple users can connect to the server and use the same web application simultaneously. Each user can also use the same application in different contexts (e.g., editing different documents). We use the term *project* to denote each such context. Each project has its own isolated web memory instance, which we call a *slot*. Projects resemble processes in traditional operating systems. Developers of traditional single-PC applications do not have to think about multiple concurrent processes, which are managed by the OS. Similarly, developers of web applications for the web computer do not have to think about multiple users working on multiple projects, since the web computer operating system (webAppOS) manages them.

2.2 The Code Space

From the web computer perspective, the code space is a pool of actions. An action, in essence, is some server-side or client-side function. We use the term *web call* to denote an action invocation or, in some cases, an action definition.

The code space relies on existing programming languages and technologies. Each action definition specifies the name of the action and how to invoke the corresponding code in the following format:

```
optional_modifiers name=instruction_set:code_location
```

The action name (before the "=" symbol) is a human-readable name, which can also be used as a reference to code. After the "=" symbol, there is a URI-like string describing the implementation of the action. The protocol part (before the ":" symbol) denotes the name of the *instruction set*, which represents a set of hardware and software requirements that may be imposed by the code. The remaining part specifies the code location.

Certain modifiers can be listed before the action name. One of them specifies *calling conventions*, i.e., the way how the arguments are passed and how the result is returned. Currently, two calling conventions are supported (they are mutually exclusive):

jsoncall the argument and the return value are encoded as JSON objects (stringified in some cases);

webmemcall the argument is passed as an object in web memory; the function returns no value, but it can modify web memory and store the results there (if any).

If the action does not require access to web memory[1], it has to be marked with the **static** modifier. If the action does not require an authenticated user session, it must be marked as **public**[2].

[1] *jsoncall* calling convention is implied for such actions.

[2] Since only authenticated users are allowed to access web memory, all non-static actions must also be non-public.

Example

```
public static jsoncall echo=staticjava:pkg.ClassName#echoImpl
```

> The action name is *echo*. The "staticjava" instruction set implies that the action is implemented as a static Java method and that a server-side Java virtual machine is required. The method name is *echoImpl*, and it is located in Java class *pkg.ClassName*. The web call can be invoked even when the user has not been authenticated (implied from the "public" modifier). Web memory will not be used (the "static" modifier). Since "jsoncall" calling conventions are specified, the *echoImpl* function must accept a JSON argument and follow other instruction set-specific conventions (e.g., in case of Java, the JSON argument will be passed as a string; the return value has to be stringified as well).

Web memory can be compared to the global variable scope. It is accessible from all non-static (and, hence, non-public) 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.

The URI part can be replaced by another URI specifying some alternative implementation of the same action. Thus, the implementation can be completely rewritten or even moved from the server to the client, or vice versa. As soon as the calling conventions and arguments match, the web call remains valid regardless of the implementation location. Thus, we say that web calls are *implementation-agnostic*.

2.3 Web Processors

Web processors are software units that are able to invoke web calls. There is usually one client-side web processor (running in the web browser) and one or more server-side web processors[3]. In some cases, remote web processors (running on remote servers) can be introduced as well. Like in traditional multi-processor and multi-core systems, developers do not need to think about which particular processor will execute a particular web call. The appropriate server-side, client-side, or remote web processor will be chosen automatically depending on the programming language and the environment required by the given web call. From the developer's point of view, the web computer resembles a multi-processor system, where web processors share the same data memory but have separate arithmetic and logic units (ALUs).

Each web processor must support at least one instruction set. Usually, multiple instruction sets available for the underlying platform are supported by a single web processor. There can be variations of instruction sets, e.g., besides generic instruction set "js" for JavaScript code, we can define also the "clientjs" instruction set for code to be executed at the client side; we can even add a

[3] There is no one-to-one mapping with physical processors.

version, e.g., “clientjs6”. Thus, instruction sets form a hierarchy based on the following “subclass of” relation definition:

> an instruction set I_2 is a *subclass of instruction set* I_1, iff code requiring environment I_1 can be executed also within environment I_2.

By convention, a web processor implementing some particular instruction set should also support its superclasses.

2.4 Web I/O Devices

Besides web memory, there can be other data sources and receivers, which we call web input/output devices. Access to some of them is standardized via APIs provided by webAppOS. Examples of such standardized devices[4] are:

- The server-side file system. This is a “cloud drive” for storing user home directories. Other remote cloud drives (such as iCloud, OneDrive, or Google Drive) can be mounted as well.
- Registry. This is a tree-like database to store user-specific and application-specific settings.
- E-mail sender. This web I/O device is useful for registering user accounts and for password recovery.
- Desktop. This device represents the web browser window and provides the ability to display standard dialogs and launch installed web applications.

Other non-standardized devices can be accessed via web calls implemented in platform-specific code using any appropriate device API for that. For instance, a client-side printer can be accessed via the JavaScript API provided by the browser. A server-side NVIDIA graphics card can be accessible via CUDA. A remote database, which can be considered a storage device, can have both server-side and client-side APIs. In case some device becomes widely used, a standardized API can be defined for it and standardized within webAppOS.

3 The Web Computer Operating System (webAppOS)

Web applications targetting the web computer do not access its main parts directly but via a set of APIs provided by the web computer OS, webAppOS. The specification of webAppOS defines:

- server-side and client-side APIs for accessing web memory;
- APIs for accessing standardized web I/O devices;
- internal APIs for drivers and services.

[4] We use the word “standardized” from the webAppOS API perspective.

Besides that, webAppOS specification defines how to deploy web applications and web services, how web applications are delivered to the end user, and how to access third-party scopes such as Google services. The specification also defines internal communication channels (buses), e.g., between the browser and the server or between web memory and web processors.

The webAppOS distribution contains out-of-the-box modules for user authentication, file system access, the desktop environment, and other services. The distribution also has the default server-side web processor implementing several instruction sets and the client-side web processor implementing the "clientjs" (client-side JavaScript) instruction set.

The main goal of webAppOS is to provide an infrastructure for web applications and web services. It also provides a uniform way to authenticate users to access webAppOS server-side resources or remote resources provided by third parties. Besides that, webAppOS factors out the execution environment. The following subsections provide the details.

3.1 Applications, Libraries, and Services

A *webAppOS application* consists of:

- a set of web calls (from the code space) implementing the business logic;
- artifacts for ensuring the communication with the end user (including delivering client-side code to the web browser).

Web calls are implemented using existing server-side and client-side technology stacks. When necessary, some exotic instruction sets can be introduced by implementing additional server-side or remote web processors having specific prerequisites installed. Some web calls can have multiple implementations for different platforms (e.g., for full-screen browsers, for mobile browsers, or for different host operating systems and processor architectures). Since web calls are implementation-agnostic, webAppOS can choose the most suitable web call implementation depending on the underlying platform.

Web calls can invoke other web calls, access web memory, access standardized web I/O devices using webAppOS APIs, access other devices using native APIs, perform computational and other tasks. Thus, the sequence of web calls resembles the execution of a classical single-PC program with traditional CALL/JUMP instructions. However, web calls can switch the execution flow between the server and the browser, which both share the same web memory state, which is being constantly and automatically synchronized.

Like classical applications, web applications can be graphical and console. We rely on HTTP to deliver graphical web applications to the end user. Similarly, we use the Web Socket protocol for delivering the output of console applications to the browser.

Classical GUI applications can be written using different technologies such as native API (Windows API or Cocoa) or using some GUI library, e.g., QT or JavaFX. Similarly, console applications can be written using native code or as

scripts (shell or Python scripts). The same applies to web applications. Graphical webAppOS applications can be deployed, for example, as HTML/JS/CSS files, PHP scripts, or Java servlets. Console web applications can be implemented using, for instance, CGI-like forwarding of input/output streams or as Java web socket servlets. To support all different ways of delivering web applications to the end user, webAppOS relies on *application adapters*.

Libraries in webAppOS resemble dynamic (shared) libraries in traditional desktop operating systems. A web library is a set of web calls that can be re-used in multiple web applications. Each web call can access the same web memory slot as used by the main web application for the current project. Web libraries are also useful for factoring out platform-specific services, where different implementations can be provided for different platforms. Thus, web applications can be developed in a platform-independent way by delegating platform-specific web calls to such libraries.

A *webAppOS service* is a module that provides useful functionality to webAppOS or third-party applications. However, end users do not access them directly. Services can be implemented as Java servlets, as client-side JavaScript code, as non-HTTP services, as Docker containers, or using some other technology. To support different service types, webAppOS uses *service adapters*.

Services can invoke web calls and access web memory, web I/O devices, and other resources. However, when being accessed, a service may require some form of user authentication as well as the context (e.g., the current project for invoking web calls).

Let us mention webAppOS WebDAV service as an example. The service provides access to the user's home directory. It uses webAppOS FileSystem API and requires users credentials (login+password) for that. The WebDAV service is implemented as a Java servlet. End users do not access WebDAV service directly, some client-side software supporting the webDAV protocol is required for that.

3.2 Scopes: A Uniform Way to Access Resources

We use the term *scope* to refer to a resource or a set of resources that require some form of authentication. Each scope has some name defined by the resource provider, e.g., Google "profile" and "spreadsheets" scopes. After successful authentication, some token is stored at the client or server side, and the resources from the desired scope become accessible by passing the token to the corresponding API.

Although the underlying resources and their APIs differ, webAppOS defines a uniform Scopes API. This API relies on scopes drivers, which perform provider-specific authentication (e.g., Google authentication via the OAuth2.0 protocol) and receive access tokens. Since authorizing scopes requires user's intervention at the client-side, Scopes API is available only at the client-side. However, scopes drivers can store tokens not only at the client side (e.g., as cookies or in *localStorage*) but also at the server side (e.g., in webAppOS Registry). Thus, webAppOS applications and services that support the APIs of the underlying resources can access them via the stored token.

For some resources, webAppOS has a standardized API, e.g., File System API for accessing resources represented as file systems. Scopes drivers should implement such standardized APIs for their underlying resources whenever possible. We call such implementations *web I/O device drivers*. By relying on these drivers, webAppOS can provide deeper integration with remote scopes and resources, e.g., by providing the ability to mount remote file systems.

Scopes drivers are implemented as webAppOS web services accessible from the client side. If a scopes driver implements some web I/O device driver, it should specify where the implementation is located (e.g., the name of the Java class that implements the File System API for the underlying scope).

Example: "google_scopes" driver

A "google_scopes" driver provides the google_scopes_driver.js script, which will be called by webAppOS whenever authentication from Google is required to access some of the Google services (e.g., "gdrive"). The google_scopes_driver.js displays the Google login window. After successful authentication, the driver stores the token in the webAppOS registry (for the given user). Besides, the "google_scopes" driver implements webAppOS File System API in some Java class (a file system driver), which takes the stored token and forwards it to Google, when the user wants to access the Google drive.

Example: "webappos_scopes" driver

webAppOS scopes driver defines the "login" scope, which displays the login page. After successful authentication, the user can access certain web I/O devices (e.g., the user's home file system) and make private web calls. The "project_id" scope extends "login". In addition, it initializes access to web memory for some webAppOS project (if the project has not been specified in the URL, the user can choose it).

3.3 Execution Environments

Typically, webAppOS runs in the *web environment*, having on or more servers that are accessed by multiple concurrent users from their browsers. By bundling the web server and the web browser component into a single desktop application, webAppOS can be launched as a standalone desktop application (we say that webAppOS runs in the *desktop environment*). If client-side code that creates graphical presentations is re-written to support small screen sizes and touch events, we can try to launch webAppOS applications in the *mobile environment*.

As a special use case, certain web applications can be created using only client-side parts of webAppOS. Such applications can rely on webAppOS client-side APIs and access third-party services, without the need to launch a webAppOS server. From the webAppOS point of view, such applications are *serverless*. Such applications can be deployed as a folder with static files that can be opened locally or served by a tiny web server.

4 Examples

In this section, we describe the steps required to create the "Hello, World!" webAppOS application and share the experience of migrating two existing applications to webAppOS, namely, OWLGrEd and DataGalaxies.

4.1 The "Hello, World!" Application

We describe how to create a simple application, where the server-side code (written in Java) stores a message in web memory and invokes a client-side web call (implemented in JavaScript) that displays that message to the end user.

A webAppOS application is deployed as a directory. It contains the webapp.properties file, where application-specific settings are specified, such as the extension for projects, application delivery type (e.g., "html" for the HTML/JS/CSS client side) and paths for finding the code (e.g., Java classpaths). The most important setting is "main", which specifies the initial web call, which will be invoked each time the project is created or opened:

```
main=HelloWorldMain
```

Since we are going to implement the main web call in Java, we declare it in the HelloWorld.webcalls file as follows:

```
webmemcall HelloWorldMain=staticjava:\
  org.webappos.apps.helloworld.HelloWorld#initial
```

According to this declaration, we have to create the HelloWorld Java class containing the static initial method:

```
package org.webappos.apps.helloworld;
...
public class HelloWorld {
  public static void initial(IWebMemory webmem, String project_id, long r) {
    ...
```

Since the HelloWorldMain web call uses the *webmemcall* calling conventions, the "staticjava" web calls adapter will pass to it the pointer to web memory, the current project id, and the reference r to some object in web memory (for initial web calls, $r = 0$).

To be able to store data in web memory, we have to define our data metamodel (e.g., in XML-based ECore syntax). Suppose we defined the HelloWorldMetamodel.ecore file containing the *HelloWorld* class having the *message* property of type *EString*. Since metamodel files are found and loaded into web memory automatically by webAppOS, the initial web call will be able to access the *HelloWorld* class right away via the *webmem* pointer[5].

However, using the web memory pointer directly is considered a low-level approach since the corresponding API resembles the assembly language. A more convenient approach is to generate Java classes that correspond to the desired web memory structure. The generator[6] can be invoked from the command line as follows:

[5] For non-Java code, a shared library for accessing web memory from Windows, Linux, and macOS native code is available.

[6] It is bundled into the webAppOS distribution.

```
../../bin/ecore2java HelloWorldMetamodel.ecore src
```

(here *src* is the target directory for Java classes).

After elevating the *webmem* pointer, we can access web memory classes as Java classes. In the following listing, we find or create a *HelloWorld* instance and set the value for the *message* property.

```
HelloWorldMetamodelFactory factory =
  webmem.elevate(HelloWorldMetamodelFactory.class);

HelloWorld objectWithMessage = HelloWorld.firstObject(factory);
if (objectWithMessage==null) {
  objectWithMessage = factory.createHelloWorld();
  objectWithMessage.setMessage("Hello for the first time!");
}
else
  objectWithMessage.setMessage("Hello again!");
```

To invoke another web call from Java, we use the server-side function *API.webCaller.enqueue*. It takes one argument, a web call seed, which specifies information about the web call.

```
WebCallSeed seed2 = new WebCallSeed();
seed2.actionName = "ShowMessageFromWebMemory";
seed2.project_id = project_id;
seed2.webmemArgument = objectWithMessage.getRAAPIReference();
seed2.callingConventions = IWebCaller.CallingConventions.WEBMEMCALL;
API.webCaller.enqueue(seed2);
```

We define the client-side *ShowMessageFromWebMemory* web call in HelloWorld.webcalls as follows:

```
webmemcall ShowMessageFromWebMemory=clientjs:helloFromWebMemory
```

Then we define the *helloFromWebMemory* function (e.g., in the script tag of index.html), which takes a web memory object as an argument and displays the message:

```
<script>
...
function helloFromWebMemory(obj) {
        alert(obj.getMessage());
}
...
</script>
```

We do not need to generate JavaScript classes (or object prototypes) to be able to access web memory from the client side—these classes (and the corresponding properties such as *getMessage*) will be created automatically on web memory synchronization.

However, to be able to use web memory at the client side, it has to be initialized via Scopes API as follows:

```
<script>
...
webappos.request_scopes("webappos_scopes", "project_id").then(
  // web-memory initialized
);
...
</script>
```

The corresponding webAppOS scopes driver will request user credentials (via the login page) and ask for the project to create or open. Then it will initialize and synchronize web memory that will become accessible as JavaScript property *webmem*.

4.2 OWLGrEd

OWLGrEd[7] is a powerful graphical editor for OWL 2.0 ontologies [2,12]. It has a high evaluation among the semantic web community [4]. Since the desktop version of OWLGrEd has been available for Windows only, we conduct an experiment of migrating OWLGrEd to the web, thus, making it accessible from multiple platforms.

First, we decided to retain the code implementing the business logic of OWLGrEd. That could not only save time but also minimize maintenance costs of existing OWLGrEd features, which must be supported in both desktop and web versions of OWLGrEd. Since the business logic code was written in Lua, we developed the "lua" web calls adapter to be used by the default server-side web processor to invoke Lua web calls. The adapter has been implemented using the LuaJ[8] library. In addition, we have created a LuaJ module for accessing web memory from Lua. This module provides the same data access API used by desktop OWLGrEd. As a result, the Lua code itself remained mostly unchanged.

However, to be able to visualize diagrams and dialog windows in the web browser, we had to re-write the corresponding graphical OWLGrEd components in JavaScript as client-side web libraries. We used the *ajoo* library for editing graph-like diagrams [14] and the DoJo Toolkit[9] for visualizing dialog windows. In addition, we used Google Web Toolkit[10] to move our layout library (for calculating coordinates of diagram elements and dialog widgets) to the web [6].

To ensure the communication between the server-side Lua code and client-side web libraries, we had to declare web calls to be triggered on certain user events (such as clicks). All technical aspects (such as data synchronization and invocations of web calls) are managed by webAppOS. Besides, webAppOS provides default dialogs for uploading, downloading, and opening projects in a way similar to opening files in classical desktop applications using a file explorer. The "Browse for file" and "Save as" dialogs are also available to webAppOS applications. Thus, OWLGrEd/webAppOS provides the same end user experience as the classical desktop-based OWLGrEd.

Finally, since webAppOS is able to synchronize web memory between multiple clients, additional clients (e.g., a debugger) can be attached to OWLGrEd. These clients can be used to manipulate OWLGrEd diagrams programmatically from the outside.

[7] http://owlgred.lumii.lv/.
[8] http://www.luaj.org/luaj.html.
[9] https://dojotoolkit.org/.
[10] http://www.gwtproject.org/.

4.3 DataGalaxies

The DataGalaxies tool provides a common space where different types of data transformations and visualizations can be joined together to perform manipulations on data and obtain the desired result. The flow of manipulations is represented graphically as a graph.

Unlike OWLGrEd, the DataGalaxies tool was initially created as a web application. However, it stored all data at the server side. Thus, when some data manipulation had to be performed at the client side, one or more round-trips were required to fetch the data from the server. When client-side code had to invoke some server-side data transformation, DataGalaxies relied on the Direct Web Remoting library, DWR[11]. The library provided a reverse AJAX implementation, which, in essence, was a patch to the HTTP protocol. The code was not elegant, but it worked.

When migrating DataGalaxies to webAppOS, we removed the DWR library and moved into web memory server-side data that had to be accessed from both the server and the browser. As a result, these data are now synchronized by webAppOS automatically via web sockets; thus, we avoid unnecessary round-trips and send data more efficiently. Besides, we removed the code that fetched data from the server. Now, data can be accessed from the client-side replica of web memory directly. As a result, the code became more elegant and more readable. We realize that if we had to develop the DataGalaxy tool from scratch, it would be much easier to implement it using webAppOS as the underlying platform. Furthermore, webAppOS applications benefit from many features available "for free" such as the default dialogs and the convenient built-in user authentication mechanism.

5 Implementation

5.1 Main Design Choices

Java is the primary language for server-side code. Since Java is platform-independent, webAppOS can be launched on a wide range of platforms. Another argument in favor of Java is that Java does not suffer from attacks based on buffer overflow. Finally, other languages can be invoked from Java using Java Native Interface, JNI, or various inter-process communication techniques.

Client-side webAppOS code is written in JavaScript as it is the de facto language for the code within the web browser. However, webAppOS does not prohibit to use other client-side technologies, which can be invoked from JavaScript[12].

[11] http://directwebremoting.org/.

[12] For instance, Java applets, VisualBasic, and ActiveX scripts can be launched by appending the appropriate tag (<script>, <object>, or <applet>) to the DOM; WebAssembly code can be launched by invoking WebAssembly.instantiate/instantiateStreaming, etc.

We use Jetty[13] as a Java-based out-of-the-box web server. At the client side, virtually any modern browser supporting JavaScript and web sockets can be used.

5.2 Implementing the Main Components

Web Memory. We use our efficient model repository AR for implementing web memory (one repository per slot) [7,8]. AR is able to use OS-managed memory-mapped files; thus, thousands of concurrent users can be served even on low-memory systems. Besides, AR uses an efficient encoding of models that resembles Kolmogorov complexity and is suitable for direct synchronization via web sockets.

The Code Space. The code space is represented by the *apps* directory at the server side. It contains subdirectories corresponding to webAppOS applications, web libraries, and services. Each subdirectory can contain **.webcalls* files containing declarations of web calls (their implementations are located in further subdirectories, e.g., *bin* for server-side Java code or *web-root* for client-side JavaScript code). Besides, there is a properties file describing how webAppOS should load, attach, and display the corresponding web application, web library, or service. Based on the data from the properties file, webAppOS finds the corresponding web application or web service adapter and registers URL paths such as /apps/myapp or /services/myservice. Web libraries are not registered, but they are loaded by webAppOS when they are required by some web application.

Client-side code is delivered according to the application or service adapter. Some adapters serve the *web-root* subdirectory; some implement redirects to local services; others implement Java servlets that generate HTTP responses on-the-fly.

Web Processors. Server-side and remote web processors are launched as separate OS processes via the corresponding *web processor adapters.* The adapters launch (or connect to) web processors and provide access to web memory. When a web processor crashes or freezes (e.g., due to some unhandled exception or an infinite cycle in a web call), the corresponding adapter can terminate and re-launch it. However, after re-launching a web processor, the underlying web memory slot is invalidated and re-loaded from the last saved state.

Typically, server-side local web processors are instances of the default web processor implemented in Java. This web processor is able to invoke web calls via the out-of-the-box web calls adapters for various programming languages such as Java and Lua.

There is only one default client-side web processor implemented in JavaScript, which relies on client-side web calls adapters for launching different types of client-side code.

Web I/O Devices. Non-standardized web I/O devices can be accessed from web calls (in the code space) via specific native APIs. However, for the standardized

[13] https://www.eclipse.org/jetty/.

web I/O devices, webAppOS scopes drivers have to be created (refer to Sect. 3.2). Scopes drivers should also contain standardized web I/O device drivers implementing the corresponding webAppOS API in Java and/or JavaScript. For the serverfull mode, only server-side web I/O device drivers are required. To support the serverless mode as well, the scopes driver should also provide an independent client-side implementation of device drivers.

After requesting and authenticating a scope, the corresponding access tokens are stored (e.g., in webAppOS registry or within client-side cookies) and can be used to access web I/O devices in that scope. Typically, these tokens are used by web I/O device drivers, which are then used internally by webAppOS to provide seamless access to the scope. However, the tokens can also be used by web calls that are able to access the scope directly.

Bridges. Both the client and the server have an internal component called a *bridge*. Bridges are responsible for:

- initializing and synchronizing web memory; the server-side bridge also manages web memory slots for different active projects;
- managing ingoing and outgoing web calls (web calls are either executed at the same side of forwarded to the other side);
- managing web processors at the corresponding network node[14].

The server-side bridge is implemented in Java as a web socket adapter for Jetty; it serves web sockets and implements server-side threads for synchronizing web memory and web calls. The client-side bridge, in its turn, is implemented via a *WebSocket* object in JavaScript.

When multiple clients are connected, the server-side bridge usually forwards client-side web calls to all of them. However, some client-side web calls can be marked as *single*. In this case, the web call will be passed only to one client, which issued the "parent" (previous) web call.

5.3 Implementing Communication Between Components

webAppOS components use several communication channels. If the communication is performed via the network or inter-process communication techniques, the corresponding channels are called *buses*. There are four main buses:

- *HTTP/AJAX Bus* is used to deliver client-side code and user interface (HTML/CSS/JavaScript) to the web browser; the bus is also used to access HTTP-based web services (including out-of-the-box services for file upload and download);
- *Web Socket Bus* is a web socket-based channel used by bridges for synchronizing web memory and forwarding web calls (when they have to be executed on the other node);

[14] Since currently there is only one client-side web processor, no manager is needed there.

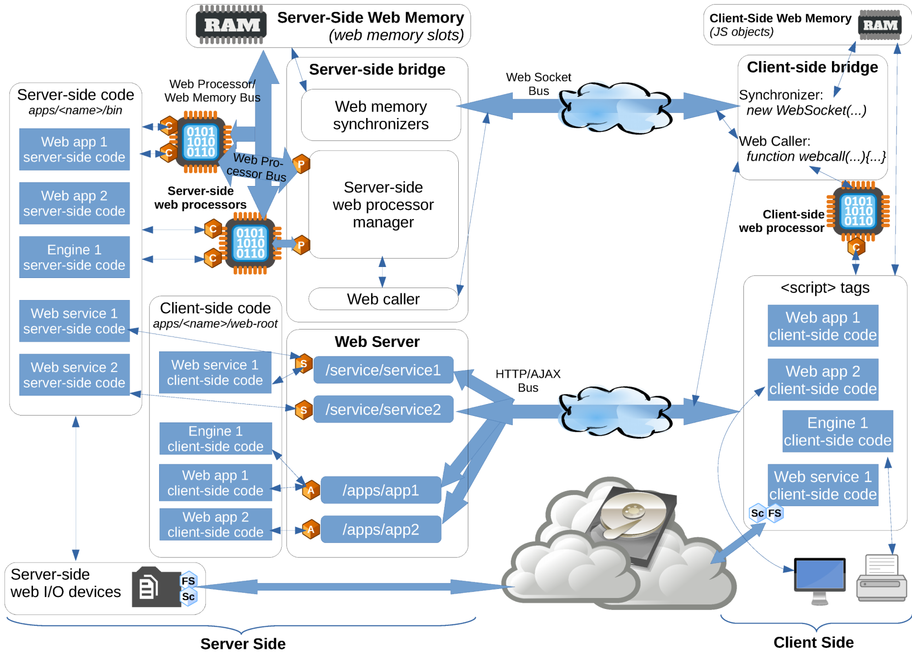

Fig. 1. Implementation of the web computer architecture (image adapted from the initial paper on webAppOS [9]). In-place communication is represented by think arrows, while buses are depicted by thick arrows. Cubes "P" and "C" stand for web processor and web call adapters, "S" and "A" for service and application adapters, "Sc" and "FS" for scopes and file system drivers.

– *Web Processor/Web Memory Bus* is used by server-side web processors (which are separate OS processes) to access web memory; the bus relies on memory-mapped files used by the AR repository;
– *Web Processor Bus* is a communication channel between the server-side bridge and server-side/remote web processors; implemented via Java RMI [13].

In-process communication is implemented via ordinary function calls or using threads. We list some examples.

– The web-server and the server-side bridge run in the same Java process; thus, they share the same memory and communicate directly. The same process is also responsible for initializing web memory slots.
– The server-side bridge also communicates directly to web processor adapters, which are implemented in Java ("P" in Fig. 1). These adapters launch (or connect to) web processors and provide them with the means to access Web Processor Bus and Web Processor/Web Memory Bus.
– The web server (Jetty) communicates directly to application and service adapters ("A" and "S" in Fig. 1). These adapters can either attach Java mod-

ules (e.g., servlets) directly or launch (or connect to) some other third-party service running on the same or remote server.
- Server-side web processors and the client-side web processor communicate with the corresponding web calls adapters directly (using Java or JavaScript calls, respectively).
- Server-side and client-side web I/O device drivers are also invoked directly by webAppOS on demand (e.g., when a mounted file system has to be accessed via a driver). However, drivers can use various communication channels internally to implement the desired functionality. For instance, the Google Drive driver will rely on Google Drive API and send requests via the network.

5.4 Addressing Scalability Issues

The webAppOS architecture described above works well on a single web server. Multiple server-side web processors can be launched to take advantage of multicore systems. Regarding web memory, we have tested 10,000 web memory slots (using AR memory-mapped files) on a single node assuming that each slot occupies just a few megabytes of RAM[15]. Thus, we are targetting to serve 10,000 concurrent connections per webAppOS server node [5].

However, to serve more concurrent users, we advise creating multiple virtual servers in the cloud. To be able to support such cloud-based deployments (each node serving approximately 10,000 users), webAppOS must be scalable. Below we explain how different parts of webAppOS can be scaled.

- Since the code space remains static for the most of time (excluding occasional configuration changes and updates), it can be shared among all webAppOS nodes as a network drive or replicated. That can be done using existing technologies (such as NFS or *rsync*).
- The server-side file system, which stores user home directories, can be shared or replicated in the same way as the code space.
- webAppOS registry can be configured to use the CouchDB no-SQL database, which has the built-in replication feature. Alternatively, the registry can be launched on a dedicated server accessible from all webAppOS nodes.
- E-mail sender (one of the server-side web I/O devices used by webAppOS itself) is specified by its URL and credentials. It can be an external server having its own load balancer, or there can be multiple local e-mail senders configured for each node individually.
- There is no need to support scalability for client-side web I/O devices since each user already has a dedicated browser instance relying on the client-side resources controlled by the user.
- We assume that one web memory slot entirely belongs to one webAppOS node. It is the responsibility of the load balancer to route web socket connections between multiple clients and multiple webAppOS nodes in a way that respects our assumption.

[15] Web-based Microsoft Office imposes the 5MiB restriction on files being edited online. Similarly, our web-based tools OWLGrEd/webAppOS and DataGalaxies normally require just a few megabytes of RAM per project.

– Adapters for stateless web applications and web services that do not require access to web memory (such as adapters serving static files or generating HTTP responses that do not depend on data from web memory) can be launched on each webAppOS node. The load balancer can switch between them randomly.
– For each stateful web application or service, only one instance will be launched on some webAppOS node. Other webAppOS nodes will redirect queries to that instance.
– For web applications and services requiring web memory, the adapters will be launched on each webAppOS node. If the request comes to the node having the required web memory slot, that node executes the request. Otherwise, the node redirects the request to the correct one.

6 Related Work

The end user experience with webAppOS resembles existing cloud-based application platforms such as iCloud, Microsoft Office Online, and Google Docs. An alternative way to communicate with the end user is the out-of-the-box webAppOS Desktop application, which provides the feeling of a classical desktop. Such web-based desktops are sometimes called "web operating systems", webOSes [10]. The term is applied mostly to client-side window managers such as Os.js[16], WebDesktop.biz[17], and AaronOS[18]. Unfortunately, they are not widely used; some of them have been discontinued (e.g., eyeOS, ZeroPC). Thus, we do not expect wide popularity of the built-in webAppOS Desktop application. Nevertheless, we can say that webAppOS is also a webOS, which goes a step further – it provides not only the client-side window manager but also the server-side environment and communication mechanisms.

There is a plethora of client-side libraries for creating rich HTML-based and single-page web applications, including AngularJS and Angular 2+[19], Dojo Toolkit and Dojo2[20], React[21], Aurelia[22], Ember[23], Vue[24], Backbone.js[25], Bootstrap[26], D3[27] as well as classical jQuery[28] and jQueryUI[29]. All they can be used

[16] https://www.os-js.org/.
[17] http://webdesktop.biz/.
[18] https://aaron-os-mineandcraft12.c9.io/aosBeta.php.
[19] https://angularjs.org/, https://angular.io/.
[20] https://dojotoolkit.org/, https://dojo.io/.
[21] https://reactjs.org/.
[22] https://aurelia.io/.
[23] https://emberjs.com/.
[24] https://vuejs.org/.
[25] https://backbonejs.org/.
[26] https://getbootstrap.com/.
[27] https://d3js.org/.
[28] https://jquery.com/.
[29] https://jqueryui.com/.

at the client-side in webAppOS (in both serverfull and serverless web applications). Different techniques to port existing non-JavaScript code to implement client-side web calls can also be used. They include Google Web Toolkit (for porting Java code)[30], Blazor (for compiling C# code to WebAssembly)[31], and others.

Popular environments that provide both client-side and server-side functionality are Node.js[32] and Meteor[33]. Unlike Node.js, webAppOS allows developers to use virtually any programming language available at the server or client side, not just JavaScript. Meteor is built on Node.js, but has a built-in client-server data synchronization mechanism, which resembles webAppOS web memory. However, Meteor uses MongoDB, a no-SQL database, which is optimized for fast queries, but not for fast writes. Besides, Meteor is also tied to JavaScript and requires to write explicit listeners in code to synchronize data, while webAppOS synchronizes web memory automatically.

Google Apps Script[34] is a platform for developing web applications based on Google services. This is an excellent choice if Google service are sufficient for the task. However, if specific server-side functionality is required, it has to be integrated manually. With webAppOS, such integration becomes easier.

CloudRail Unified APIs[35] was an initiative to provide universal APIs for various cloud services. It resembled standardized APIs for scopes and webAppOS web I/O device drivers. Regretfully, the unified API branch was discontinued by CloudRail on March 1, 2019. We hope that webAppOS devices can take over the baton by providing free and open-source APIs and implementations of the corresponding drivers for different cloud service providers.

An interesting approach for bringing traditional desktop applications to the web is via cloud platforms such as RollApp and AlwaysOnPC, where windows of classical applications are forwarded to the web browser[36]. Open-source libraries such as Gnome Broadway[37] and xpra[38] as well as commercial Citrix Virtual Apps[39] use a similar approach. The same approach can be introduced in webAppOS. However, it would require a dedicated web processor and more RAM for each running application; thus, the number of concurrent users that can be served simultaneously would decrease significantly.

Although we rely on our model repository AR to implement web memory, linked data and semantic web technologies such as RDF and OWL could also be used for that [18–21]. Semantic reasoners can even be viewed as specific instruction sets for web processors [3].

30 http://www.gwtproject.org/.
31 https://blazor.net/.
32 https://nodejs.org/.
33 https://www.meteor.com/.
34 https://www.google.com/script/start/.
35 https://cloudrail.com/.
36 https://www.rollapp.com, http://www.alwaysonpc.com/.
37 https://developer.gnome.org/gtk3/stable/gtk-broadway.html.
38 https://xpra.org.
39 https://www.citrix.com/products/citrix-virtual-apps-and-desktops/.

The Electron[40] framework is intended to simplify the development of cross-platform desktop applications using web technologies. That resembles how webAppOS is intended to support multiple target environments (web, desktop, and mobile), but without the requirement to use JavaScript for all the code.

There is an interesting relation between the web computer and the architecture of classical computers. If we re-arrange main elements from Fig. 1, we come up with Fig. 2(a), where we can notice a similarity with the typical motherboard layout (Fig. 2(b)).

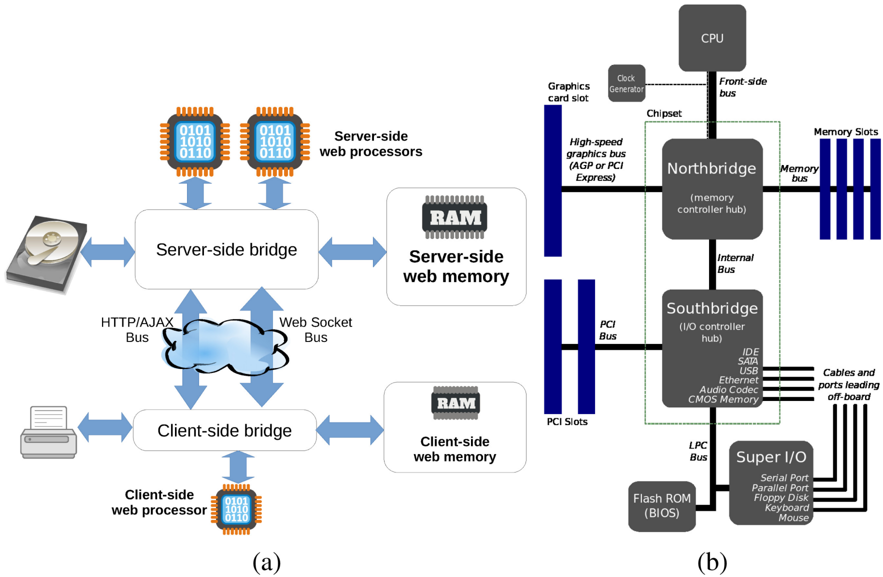

Fig. 2. (a) The overall webAppOS architecture. (b) A typical layout of the north and south bridges (image by Gribeco and Moxfyre, CC BY-SA 3.0).

7 Conclusion

The main advantage of using webAppOS to develop web applications is the illusion of a single target computer. This illusion corresponds to the physiology of the human brain; thus, web applications can be created faster and at a higher level of abstraction, where the network is factored out. The webAppOS learning curve is also very straightforward. Another benefit is that webAppOS provides common grounds for virtually all types of web applications and services. By using the appropriate adapters and drivers, different parts of the web application can be written using different technologies and programming languages. Since

[40] https://electronjs.org/.

webAppOS is open-source[41], it facilitates the usage of private web servers, where the users have more control over their data [15].

Technical strengths of webAppOS are the presence of automatically and transparently synchronized web memory and the ability to invoke code via web calls in an implementation-agnostic way. Synchronization is very fast. It bases on web sockets and efficient model encoding provided by the AR repository.

Alan Kay, a Computer Science pioneer, once said that the web browser acts as a mini-operating system. We would say that webAppOS is a step further; it is a superstructure over *both*, the server-side OS and the client-side web browser. Perhaps, webAppOS can eventually become standardized "kernel" for existing diversified web applications and services, similarly how Linux became a *de facto* kernel for GNU software [17]. However, significant efforts from the open-source community, as well as support from existing cloud service providers and other parties, are required for that.

Someone may ask: Why Google will not do the same? In fact, Google has a platform used by Google Docs. However, their platform is not open-source. Besides, most cloud services (including Google) rely on existing, proven technologies, where the learning curve might be longer, but is more predictable. We can say that webAppOS is the inversion of the Google approach. On the one hand, the web computer metaphor is closer to the human brain, but the underlying platform is new and not widely used at the moment. Nevertheless, it is innovative and open. We hope that webAppOS will be useful for both the open-source community as well as for developers of commercial web applications.

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/VI-AA/1/16/214 "Model-Based Web Application Infrastructure with Cloud Technology Support".

References

1. Andrews, M., Whittaker, J.A.: How to Break Web Software: Functional and Security Testing of Web Applications and Web Services. Addison-Wesley Professional, Boston (2006)
2. Barzdins, J., Barzdins, G., Cerans, K., Liepins, R., Sprogis, A.: OWLGrEd: a UML style graphical notation and editor for OWL 2. In: Proceedings of OWLED 2010 (2010)
3. Corno, F., Farinetti, L.: 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 (2012). http://elite.polito.it/files/courses/01LHV/2012/7-OWLreasoning.pdf
4. Dudáš, M., Lohmann, S., Svátek, V., Pavlov, D.: Ontology visualization methods and tools: a survey of the state of the art. Knowl. Eng. Rev. **33**, e10 (2018). https://doi.org/10.1017/S0269888918000073
5. Kegel, D.: The C10K problem. http://www.kegel.com/c10k.html

[41] http://webappos.org.

6. Kozlovics, S.: Calculating the layout for dialog windows specified as models. In: Scientific Papers, University of Latvia, vol. 787, pp. 106–124 (2012)
7. Kozlovičs, S.: Efficient model repository for web applications. In: Lupeikiene, A., Vasilecas, O., Dzemyda, G. (eds.) DB&IS 2018. CCIS, vol. 838, pp. 216–230. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-97571-9_18
8. Kozlovičs, S.: Fast model repository as memory for web applications. Databases Inf. Syst. X **315**, 176–191 (2019)
9. Kozlovičs, S.: The web computer and its operating system: a new approach for creating web applications. In: Proceedings of the 15th International Conference on Web Information Systems and Technologies (2019). https://doi.org/10.5220/0008053800460057
10. Lawton, G.: Moving the OS to the web. Computer **41**(3), 16–19 (2008). https://doi.org/10.1109/MC.2008.94
11. Object Management Group: OMG Meta Object Facility (MOF) Core Specification Version 2.4.1 (2011)
12. Ovčinnikova, J., Čerāns, K.: Advanced UML style visualization of OWL ontologies. In: Proceedings of the Second International Workshop on Visualization and Interaction for Ontologies and Linked Data co-located with the 15th International Semantic Web Conference (ISWC 2016) CEUR 1704, pp. 136–142 (2016)
13. Pitt, E., McNiff, K.: Java.Rmi: The Remote Method Invocation Guide. Addison-Wesley Longman Publishing Co., Inc., Boston (2001)
14. Sprogis, A.: ajoo: WEB based framework for domain specific modeling tools. In: Frontiers in Artificial Intelligence and Applications Volume 291: Databases and Information Systems IX (2016)
15. Stallman, R.: Who does that server really serve? (2010). http://www.bostonreview.net/richard-stallman-free-software-DRM
16. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Boston (2008)
17. Tozzi, C.: For Fun and Profit: A History of the Free and Open Source Software Revolution. The MIT Press, Cambridge (2017)
18. W3C: OWL Web Ontology Language reference (2004). http://www.w3.org/TR/owl-ref/
19. W3C: OWL 2 Web Ontology Language Document Overview, 2nd edn. (2012). http://www.w3.org/TR/owl2-overview/
20. W3C: RDF Vocabulary Description Language 1.0: RDF Schema (2014). http://www.w3.org/TR/rdf-schema/
21. W3C: Resource Description Framework (2014). http://www.w3.org/RDF/