

The *webAppOS* Architecture

Sergejs Kozlovičs

Research #1.1.1.2/VIAA/1/16/214 “Model-Based Web Application Infrastructure with Cloud Technology Support”, project agreement #1.1.1.2/16/I/001.

This document is marked as Deliverable 2.1 “The description of the architecture of the proposed infrastructure” within Working Package 2 “Defining the specification of the infrastructure”.

Abstract

The purpose of this document is to describe the overall architecture of webAppOS as well as design choices.

This document references the API specifications (Deliverable 2.2) and other webAppOS documentation (Deliverable 3.2), where further details are provided.

The latest version of this document can be obtained at <http://webappos.org/theory>.



Contents

1	Introduction	4
2	The Web Computer Architecture	5
3	Web Memory: Storing and Accessing Data	6
3.1	The Main Design Choice	6
3.2	Accessing Data in Web Memory	7
3.2.1	Accessing Web Memory at the Server Side.	9
3.2.2	Accessing Web Memory at the Client Side.	9
3.3	Multiple Web Memory Instances on a Single Server	9
3.4	Collaboration	11
3.5	Implementing Web Memory	11
3.6	Code References	12
4	Web Calls: Launching Code	12
4.1	Instruction Sets	12
4.2	Web Calls	13
4.3	Web Calls Adapters	13
5	Web Processors	15
5.1	The Client-Side and Server-Side Bridges	15
5.2	Implementing Client-Side Web Processors	16
5.3	Implementing Server-Side Web Processors	16
5.4	Implementing Remote Web Processors	17
5.5	Web Processor Bus Service	17
6	Web I/O Devices	17
6.1	Scopes	18
6.1.1	Scopes API	18
6.1.2	webAppOS Scopes Driver	19
6.2	Predefined Web I/O Devices	19
6.2.1	webAppOS Registry	19
6.2.2	webAppOS Home File System	20
6.2.3	E-mail Sender	21
6.2.4	Desktop	21
6.3	Mounting Remote File Systems and Registries	21
6.3.1	Remote (Cloud) File System Drivers	21
6.3.2	Remote (Cloud) Registry Drivers	22

7	webAppOS Applications and Services	22
7.1	webAppOS Web Applications	22
7.2	webAppOS Engines	23
7.3	webAppOS services	23
7.4	Serverfull and Serverless Modes	24
8	Overall webAppOS Architecture	24
8.1	Main Components	25
8.2	Similarity with the Typical Motherboard Layout	26
8.3	Developers' Experience	27
8.4	End-Users' Experience	27
8.5	Administrators' Experience and Scaling	27
	References	28

1. Introduction

When Ada Lovelace wrote programs for the Babbage engine, there was nothing more than a single computer (engine) operated by a single user and being able to run one program at a time. Later, when Alan Turing and Emil Post described the Post-Turing machine, the same way of thinking remained: a single user operated a single computer being able to execute a single program at a time. We call such thinking *the three-singles assumption*.

Today we have multi-core processors and advanced operating systems with support for multitasking, multiple user accounts, and networking. Nevertheless, programmers today still tend to preserve the same level of thinking as in the three-singles assumption, leaving other technical tasks to the operating system, OS, which deals with multitasking, user management, and concurrent access to the computer resources (CPU, memory, and I/O devices).

The urge to preserve the 3-singles assumption in programmers' mindset is not illogical. Recent neuroscience research reveals that the human brain is not capable of real multitasking; it switches between tasks instead.

Modern operating systems are intended to be executed on a single computer, where all main resources (CPU, memory, and I/O devices) are physically attached. This design can be traced back to the architectures of the first general-purpose electronic computers. The most known architectures are the von Neumann architecture (also called Princeton architecture) and the Harvard architecture [23]. The main distinction between them is that in the von Neumann architecture, instructions and data are stored in the same memory, while the Harvard architecture uses 2 types of memory for that. The von Neumann architecture is easier to implement and is more suitable for code loaders and just-in-time compilers. The Harvard architecture, in its turn, allows instructions and data to be processed in parallel and even to be stored using different physical representations. While the majority of today's computers, in general, follow the von Neumann model, modern CPUs also implement some aspects of the Harvard architecture in their cache for optimized performance.

Both architectures describe a single computer. That is not surprising since both architectures appeared before Vannevar Bush foreshadowed the modern Internet in his classical paper [24]. Introduction of the network required no changes in the architecture since the network could be represented by I/O devices (such as network cards or antennas) attached to the communication endpoints. However, writing network programs required a new way of thinking: the programmers had to think about multiple computers (such as the client and the server), where the resources are physically separated. Operating systems could only aid with providing seamless access to some remote I/O devices (such as network printers, remote displays, or remote file systems), but failed to provide the same indirection level to remote processors and memory. Furthermore, web applications had to address additional network-specific issues such as managing limited server resources among multiple users, network latency, connectivity, security, and privacy.

webAppOS (abbreviation from *Web Application Operating System*) is an infrastructure that allows the developers to create web applications while remaining within the 3-singles assumption. *webAppOS* factors our most network-related technical issues and provides an illusion of a single computer with directly attached CPUs, memory, and I/O devices. We coin the term *web computer* to denote this illusion. The web computer is an abstraction since it exists only in the mind of the developer. The *webAppOS* API provides a feeling that the underlying web computer is real. Still, *webAppOS* must rely on existing web

technologies for delivering web applications to and running them at real devices.

In Sections 2–6, the architecture and the main internal components of the web computer are described. Section 7 provides more detail on how to develop webAppOS applications and services. Section 8 provides a summary of the webAppOS architecture.

2. The Web Computer Architecture

Like in classical computer architecture, the hypothetical web computer has 3 main types of components: *web memory*, *web processors*, and *web I/O devices*. We use the traditional client-server architecture as a starting point to explain these 3 terms.

Traditionally, the web server usually runs some HTTP server software (e.g., Apache web server) and is able to execute some server-side application code (e.g., PHP code or Java servlets). There is also the client-side web browser being able to display HTML pages and run JavaScript¹ or WebAssembly code. Traditionally, the communication between the client and the server is performed via stateless HTTP requests/responses using HTML, XML, or JSON data formats. Modern web applications can also use web sockets for lightweight bi-directional communication.

To provide an illusion of the common memory space for data (not for the code!), the network communication has to be factored out. Two steps are required for that:

- defining a standard data format for both the server and the client,
- synchronizing these data transparently and automatically.

We call such common data space *web memory*.

An essential property of web memory is that it stores data only, the code is stored separately. Thus, the web computer implements the Harvard architecture. The arguments in favor of such design choice are:

- Security considerations. Should web memory store instructions along with data, the client would be able to inject malicious code that could be eventually executed at the server side.²
- Code concealment. Due to license restrictions or copyright-related concerns, it can happen that specific proprietary server-side code may not be disclosed. On the other hand, the client can have some proprietary client-side code (e.g., encryption algorithms) that should not be disclosed to the server.³
- Different environments. There are fundamental distinctions between the server-side and client-side environments and sets of technologies (e.g., PHP/Java/Python at the server-side and JavaScript/WebAssembly at the client-side). On the contrary, storing code and data in the same place would require unification of code execution

¹We use the name “JavaScript” here, however, the language standard is called ECMAScript [25].

²It is possible that in the future, web memory could allow instructions to be stored in the data space, but then those instructions must be executed in a controlled environment. That would resemble how web browsers restrict client-side JavaScript code from accessing foreign iframes as well as the local file system but implemented at the server-side.

³If such code needs to be shared with the server via the data space, it has to be obfuscated first, e.g., using of homomorphic encryption [26].

environments at both ends.⁴ Besides, by keeping the code separate from data, we can rely on existing classical techniques for delivering the code between the server and the client (e.g., the HTML *script* tag for delivering JavaScript, or some FTP/NFS directory for delivering executable files in virtually any format).

- Web memory is a precious resource. Since multiple users can connect to the server simultaneously, server RAM and other resources used to implement web memory should not be wasted.

Web-processors are server-side and client-side software (not hardware!) units being able to launch different types of code written in traditional programming languages. For instance, there can be a web processor running Java VM at the server side and the web processor running JavaScript interpreter in the browser window at the client side. There is no bijection between web processors and physical processors (cores). For instance, one physical processor at the server side can be used to run 2 web processors: the PHP interpreter and the Java Virtual Machine (JVM). On the other hand, the same JVM web processor can launch multiple threads, which could be executed within multiple physical processors or cores.

Sometimes data have to be sent to/received from external data stores such as a server-side database or cloud storage. We use the input/output device metaphor to denote such external data sources and receivers (we call them *web I/O devices*). Specific graphical presentations within the browser window (e.g., the web desktop) and client-side devices (such as printers) are also considered web I/O devices by webAppOS.

Figure 1 provides a summary on the web computer architecture by depicting its main components.

The following sections provide a detailed, in-depth explanation for all the main components of the web computer.

3. Web Memory: Storing and Accessing Data

3.1. The Main Design Choice

The main design choice regarding storing data in web memory is: data are represented as a MOF-like model, not as an array of bytes. MOF (Meta-Object Facility) is an OMG standard for describing models [27]. A model, in essence, is a graph of objects and links conforming to some specification (language) called a metamodel. Metamodels, in their turn, correspond to some universal meta-metamodel, which can describe itself. Since various technologies⁵ converged to the same principles as used within MOF-like models, we use the universal model concept to represent the content of web memory.

The arguments in favor of such second design choice are:

- Models and metamodels are easy formalized. MOF and ECore are the most known modeling standards based on the same principles, but using slightly different meta-metamodels. Since models must conform to some metamodel, which must conform to the chosen meta-metamodel, it is easy to perform memory consistency checks and

⁴Node.js tries to provide the same environment for both the client and the server, but this imposes restrictions to the programming language (JavaScript only).

⁵These are technologies using 3 meta levels, e.g., object-oriented programming languages, XML, databases, ontology languages, etc. J. Bézivin and I. Kurtev used to call them *technical spaces* [28, 29].

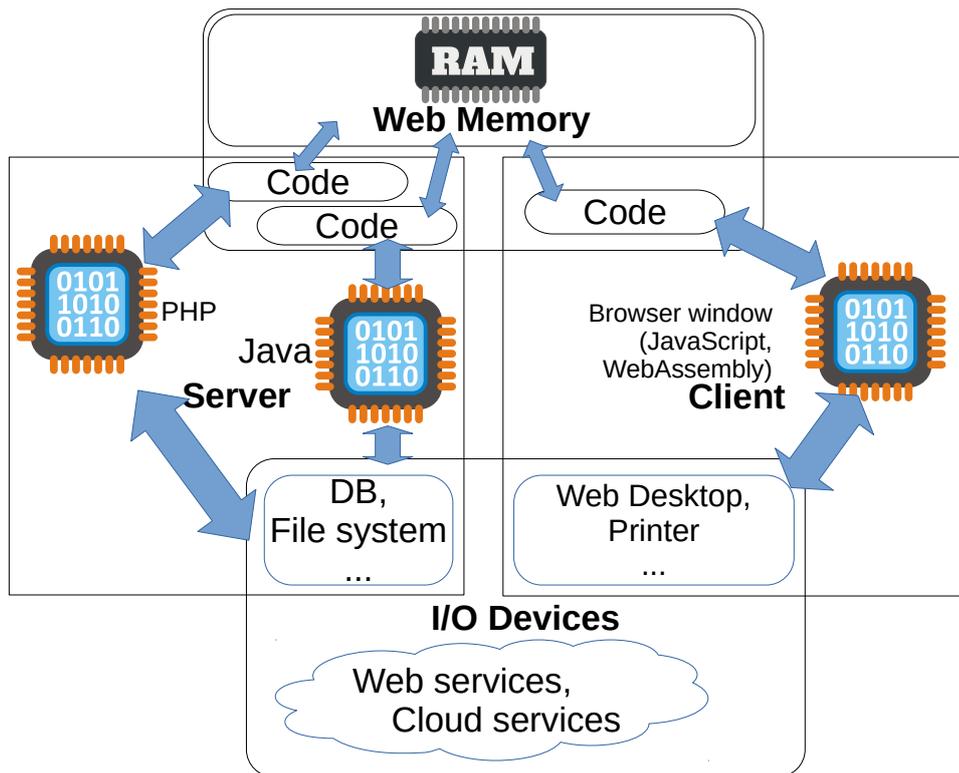


Figure 1: The web computer architecture: the first approximation. Processor icons denote web processors.

provide additional functionality such as granular undo/redo mechanism for objects-based web memory [30].

- The similarity to OOP. Programmers are accustomed to object-oriented programming (OOP). Web memory objects can be mapped the corresponding structures of the chosen OOP-style language (including JavaScript), thus, by generating specific OOP wrappers, developers can use web memory objects as if they were native objects.
- Easier synchronization. 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. In contrast, array-based web memory would require a round-trip delay and server-side management for each memory block allocation to avoid collisions.
- The same approach is used in Java. In Java, memory is also a graph of objects conforming to the Java specification. That allows the Java Virtual Machine to control memory management (e.g., to implement the automatic garbage collector) and provide the reflection mechanism.

3.2. Accessing Data in Web Memory

Figure 2 depicts various ways of how data stored in web memory can be accessed from code. Figure 2 expands “Code↔Web Memory” arrows from Figure 1.

The main API to access web memory is RA-API (Repository Access API), borrowed

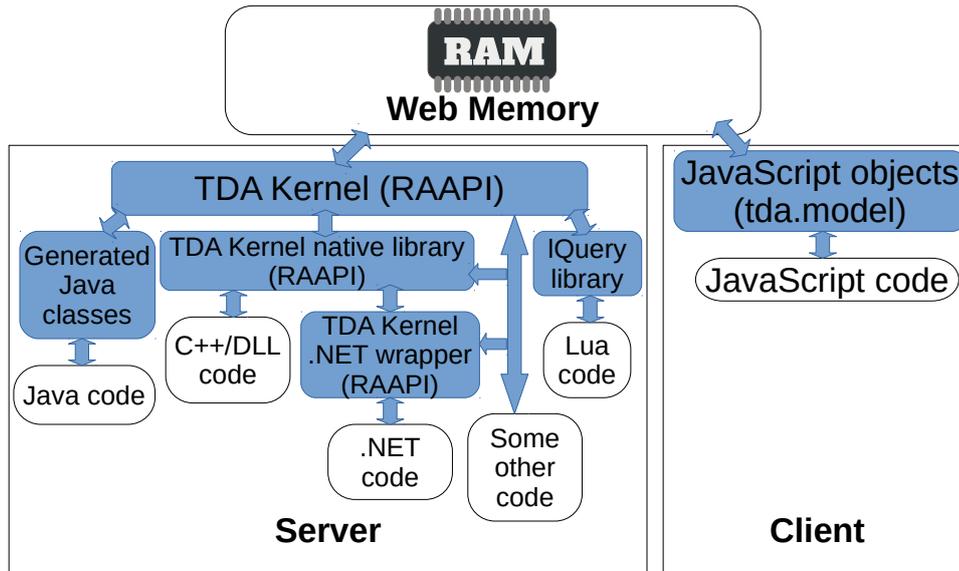


Figure 2: Accessing web memory from code.

from the Transformation-Driven Architecture, TDA.⁶ The reasons for choosing RAAPI are:

- It is compatible with virtually any model storage. In particular, there is a RAAPI wrapper for ECore [33].
- It is a low-level API. RAAPI can be viewed as an assembler for creating and traversing models; thus, RAAPI is able to access model storage efficiently. On the contrary, high-level APIs (such as *Epsilon Model Connectivity Layer* or *ATL Model Handler Abstraction Layer*), are not so efficient (e.g., linked objects can only be set as a list, even when we need to include/exclude just one object). Besides, they conceal internal data structures too much.
- RAAPI supports multiple ontological meta-levels, while most operations reside in two meta-levels. That is in accordance with Šostaks' conjecture [34]:

*It is difficult for a human to think at more than two meta-levels at a time.
Still, it is fairly easy for a human to focus on any two adjacent meta-levels.*

Besides RAAPI, we also borrow the TDA Kernel component, which implements RAAPI. TDA Kernel is able to open different model repositories via different repository adapters. We extended TDA Kernel with additional functionality such as the ability to synchronize RAAPI actions. TDA Kernel is written in Java, but it comes with native wrapper for Linux, Windows, and MacOS as well as a .NET wrapper.

An important feature of TDA Kernel is the ability to call external code when links of certain kinds are created. TDA Kernel defines 3 special classes for that: `TDAKernel::Event`, `TDAKernel::Command`, and `TDAKernel::Submitter`. The latter is a singleton

⁶We proposed RAAPI in 2013 by combining the best from existing repository APIs. RAAPI can be mapped to virtually any model repository. The actual version can be found at <http://webappos.org/dev/raapi/>. The Transformation-Driven Architecture, TDA, is a model-driven approach for building desktop tools. We proposed it in 2008 [31]. In 2013 we proposed a more mature version 2 of TDA [32].

object, to which events and commands can be linked. Particular event and command types are descendants of `TDAKernel::Event` and `TDAKernel::Command`. When a command or an event object is linked to the singleton submitter object, TDA Kernel passes the control to the event/command hook, which calls the corresponding external code and passes the event or command object to it. The main distinction between events and commands is as follows: each command of the same type will call the same code, but events can cause different code to be executed depending on registered event listeners.⁷ The TDA event/command mechanism allows model transformations to call external code by using only manipulations with the model; thus, the calls become environment-agnostic. `webAppOS` will support TDA calls as one of the ways to call code (discussed in Section 4.2).

3.2.1. Accessing Web Memory at the Server Side.

One option to access web memory at the server-side is via RAAPI by using either Java-based TDA Kernel or some of its wrappers (native or .NET). However, this should be considered low-level memory access, since RAAPI covers only primitive operations on models.

Another option is as follows. Given a metamodel that describes the structure of web memory, we can generate Java, C++, or other programming language classes that map model classes to the corresponding language constructs. That could allow the programmers to access all objects in web memory as if they were created natively. However, for the majority of languages, we cannot modify classes dynamically, thus, this trick works only with a static metamodel, where classes are fixed, and only their instances change.

One more option is to develop higher-level languages providing more flexible access to the model. For instance, the `IQuery` library is a RAAPI wrapper that provides `xpath`-style language for accessing the model from the Lua programming language [35].

3.2.2. Accessing Web Memory at the Client Side.

In most cases, synchronized web memory at the client side will be accessed from JavaScript code.⁸ Since JavaScript is an interpreted language, we can dynamically (during synchronization) create JavaScript object prototypes that correspond to classes in the model. Then, for each object in the model, we create a JavaScript wrapper object. Thus, programmers can have the feeling of working with native JavaScript objects, while all changes are being transparently synchronized with the server. For client-side JavaScript code, no RAAPI is required at the client side.

JavaScript wrapper objects are accessible via the `tda.model` object after including the `webappos.js` script [1].

3.3. Multiple Web Memory Instances on a Single Server

Since there usually are multiple connections to the server, `webAppOS` should isolate their web memory. Each user can work with multiple web applications at the same time. In some instances, the user can work with the same web application in different contexts (e.g., editing different documents). We use the term *project* to denote each such context. We use the term *slot* to denote a web memory instance used by a particular project.

⁷Event listeners are usually registered somewhere in the context of the event object, e.g., in the `on<EventName>` attribute of some object linked to the event. TDA Kernel is able to collect all such event listeners for the given event.

⁸API for accessing web memory from WebAssembly code is subject to further work.

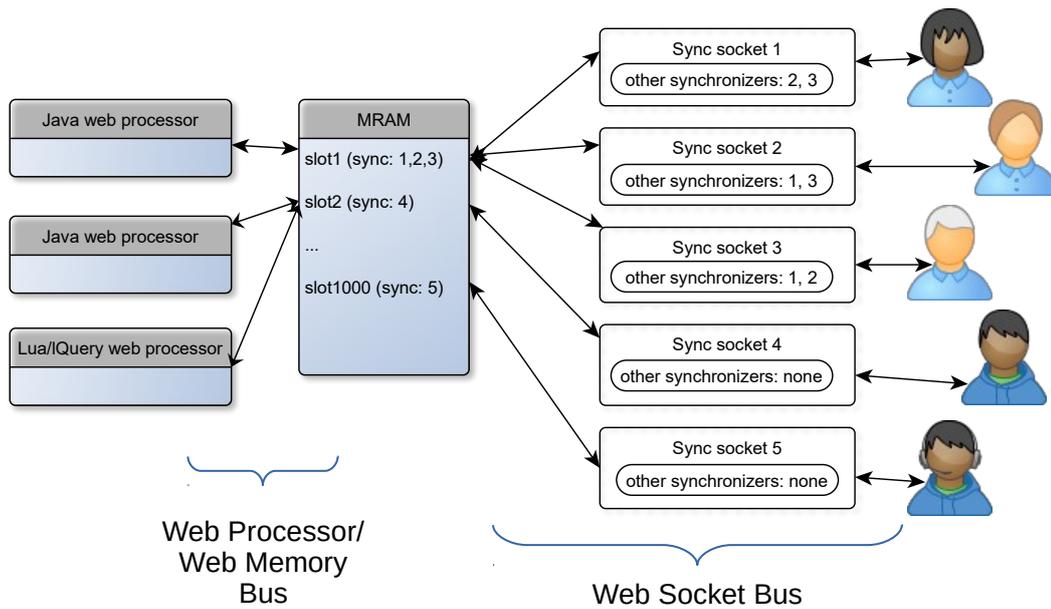


Figure 3: MRAM slots and corresponding buses.

We use the RAM⁹ homophone *MRAM*¹⁰ to denote the server-side manager for all web memory slots.

MRAM slots are located at the server. Each MRAM slot can be accessed either from the server side (by server-side code) or from the client side (by client-side code), see Figure 3.

MRAM slots must be independent on each other, since when web memory in one slot is invalidated (e.g., due to a code exception during server-side code execution or due to a client-side connection lost), other slots must remain valid. The invalidated memory slot can be restored later from the last saved state.

To keep MRAM slots safe from potential server-side exceptions, webAppOS launches server-side web processors as separate OS processes. Thus, if some web processor process terminates unexpectedly (e.g., due to an exception), we can safely invalidate the corresponding memory slot and re-launch the web processor for further tasks. The communication channel between web processors and MRAM slots is called *Web Processor/Web Memory Bus* [2].¹¹

To be able to access web memory from the client side, a synchronizer, which syncs the content of the web memory slot with the client browser, is attached to the TDA Kernel of that slot. The synchronization channel between the server and the clients is called *Web Socket Bus* [3].

⁹Random Access Memory, an abbreviation used to denote memory in traditional computers.

¹⁰Model Repositories as Memory. It is implemented as a pool of model repositories, as explained in Section 3.5.

¹¹We use the bus metaphor to denote a communication channel between different OS processes or between different network nodes.

3.4. Collaboration

If multiple users are collaborating on the same project, the same MRAM slot is synchronized via multiple synchronizers.¹² In the case of multiple synchronizers, the changes coming from user i are synced between all users except i .¹³ The changes coming from server-side web processors are synced between all users connected to the same slot. However, this only ensures the consistent state of web memory, not graphical presentations, which have to be updated by other means.

3.5. Implementing Web Memory

Although web memory could be implemented in a (graph) database, the following considerations lead us to use a model repository as a more efficient solution:

- There is a need for fast web memory synchronization, both initial and incremental. Traditional string-based serialization techniques are usually slow.
- Server-side and client-side should be able to alter web memory independently, without a round-trip delay (however, their changes must be eventually merged into a consistent state).
- Web memory has to be optimized for a *large number of primitive* operations used to navigate through the model. In contrast, databases are usually invoked for executing *infrequent*, but *compound* queries.
- There is no need for ACID¹⁴ properties since web memory acts as RAM analog.
- Programmers should take care of memory consistency (like in traditional single-PC applications).

AR is a multi-level model repository that was developed with the goal to provide web memory to webAppOS [36, 37]. Each MRAM slot is backed by an AR instance. AR has the following features:

- AR implements RA-API and, thus, can be accessed via TDA Kernel.¹⁵
- AR uses the efficient encoding of models. The encoding is optimized for frequent primitive operations.
- AR is designed for efficient synchronization via web sockets. The encoding of models suitable for transferring via web sockets “as is” (in particular, it uses IEEE doubles for numbers, since it is the only numeric datatype available in JavaScript). Changes are sent asynchronously; thus, the primary process continues without any delay. Furthermore, changes are sent via web sockets in bulk. Besides, the repository is able to merge independent server-side and client-side changes. The following trick

¹²Notice that this ensures only memory state synchronization. Synchronizing graphical representations, such as diagrams, depends on the logic of web applications, thus, lies behind webAppOS features. Nevertheless, webAppOS will make it easier to create fully collaborative web applications by offering certain helper functionality.

¹³We use efficient data structures for lists of synchronizers; thus, for n users, the memory occupied by these lists is $O(n)$.

¹⁴Atomicity. Consistency. Isolation. Durability.

¹⁵AR is implemented as a repository adapter for TDA Kernel.

is used to avoid collisions: when new repository elements are created, the server assigns even references for them, but the client assigns odd¹⁶.

- AR is able to utilize memory-mapped files, managed by the OS. This allows AR to serve 10,000 and more web memory slots used by concurrent users, even when all the slots do not fit into server physical RAM. Still, due to possible paging and limited processor cache, web memory should be considered a precious resource and limited to a few megabytes per slot in popular server configurations.¹⁷ Since each AR instance has a unique identifier for memory-mapped files, when one MRAM slot corresponding to the given AR instance is invalidated, other MRAM slots remain valid. During the invalidation process, AR deletes memory-mapped files; thus, when the repository is being accessed next time, it is re-created from the last saved state.

3.6. Code References

While web memory does not store code, it is allowed to store *code references*. A code reference is just a human-readable string identifier, which maps to a particular server-side or client-side code (e.g., function). Since the client can modify code references within web memory, webAppOS will execute only code pointed by code references registered at the server side. That is required to avoid code injection-based attacks. The referenced server-side code should have been validated that it will not harm the server.

4. Web Calls: Launching Code

Since the code of a web application can be developed for different execution environment, we define the term “instruction set” in the context of the web computer architecture first. Then we explain how that code is called.

4.1. Instruction Sets

The code can be located either at the server side or at the client side. The server and the client usually have different environments and technology stacks. We use the term *instruction set* to denote each such environment. An instruction set can be represented by (but not limited to) the following environment types:

- a particular OS regardless of the processor architecture, e.g., “Ubuntu 18.04”, meaning that the Ubuntu Linux v18.04 is installed and certain packages are available, however, the processor architecture may vary (e.g., Intel or ARM);
- a combination of the OS and the physical processor architecture (e.g., “GNU/Linux-x64” or “Win32”);
- a higher-level technology (e.g., PHP, Java, Python, .NET, JavaScript, WebAssembly, etc.);
- presence of certain software or hardware (e.g., “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:

¹⁶In case there are several servers or clients, we can further split the sets of references modulo 4, 8, etc.

¹⁷For example, Microsoft Office 365 limits the size of one document to 5 MiB.

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

Example. Assume the “JavaScript” instruction set has two subclasses, “Server-side JavaScript” and “Client-side JavaScript”. The algorithmic code can require just “JavaScript”, allowing it to be executed by webAppOS at either side. However, GUI code must be executed in a web browser, thus, it should require the “Client-side JavaScript” instruction set. Further subclasses, including JavaScript version numbers, can also be defined.

Since the code will be executed within web processors, each web processor must support at least one instruction set. It is up to a web processor to which extent it will support the hierarchy of instruction sets. Typically, a web processor implementing some particular instruction set should also support its superclasses (see Section 5).

4.2. Web Calls

When a project is being created or opened by the user, an MRAM slot is allocated, and the initial code (the initial action) of the corresponding webAppOS web application is called (the initial action plays the same role as the main function in C/C++ or Java). Initial action can make calls to other actions (client-side or server-side), which can call other actions, and so on. Thus, we can consider a webAppOS application as a set of such actions, which may require different environments (instruction sets) to run. Some of these actions will be defined by developers of a particular web application, some are provided by webAppOS out-of-the-box, others will be available as pluggable libraries. We use the term *web call* to denote a single invocation of a webAppOS action.

Each action has a name (also used as a *webAppOS code reference* mentioned in Section 3.6), which uniquely identifies the given action. Each action name maps to the associated information containing the name of the required instruction set, where to find the corresponding code, how to pass the arguments, as well as some other technical information. We call all this information *web call declaration*.

Each invocation consists of the action name and the required arguments. All registered actions can be called from the server side as well as from the client side. Thus, when making a web call, the developer does not need to know where and how the code will be executed.¹⁸ Given a web call, webAppOS searches for the corresponding action by its name. Then, webAppOS finds the appropriate web processor that supports the required instruction set and forwards the call to that web processor (the web call is serialized and sent over the network if the appropriate web processor is running on another network node).

4.3. Web Calls Adapters

Web calls adapters act as an abstraction layer for invoking code within different environments (instruction sets). Each web calls adapter implements a uniform server-side or client-side API (web processors rely on these adapters when processing web calls) [4, 5]. Examples of existing adapters are:

- *staticjava* - a server-side adapter for calling Java code located in static Java functions (thus, the adapter just has to load the corresponding Java class, but does not need to create a class instance for each new call);

¹⁸This resembles how traditional operating systems choose the processor for executing the code, where the developer does not need to think about which processor or core will be used.

- `lua` - a server-side adapter for calling Lua 5.1 code;
- `clientjs` - a client-side adapter for executing JavaScript code.

Additional web calls adapters can be created as well.

Each web call adapter has a name, which corresponds to the instruction set. If the name contains the substring “client”, it is considered a client-side web call adapter (server-side, otherwise).

We use the term *calling conventions* to denote different ways of how arguments can be passed with a web call. Currently, there are two calling conventions:

- **jsoncall** - a string or a JSON is passed as an argument; a JSON is always returned. Internally, when performing a call or when sending JSONs via the network, JSONs are serialized (stringified). This calling convention is used for actions that do not require access to web memory as well as for actions that can access web memory, but where the argument should not be stored in web memory, or it is easier to encode the argument as JSON.
- **tdacall** - a web memory object (or its reference) is passed as an argument, nothing is returned. This calling convention is mostly used to execute actions that access the model. In particular, `tdacall` is used for executing TDA commands and handling TDA events. All `tdacall` actions are executed asynchronously; thus, they are expected to return nothing. If they need to provide some feedback, they can do that, for example, by creating and submitting a TDA event, which will be handled in a similar way.

Each web calls adapter must implement at least one of the calling conventions.

Each web call declaration must specify the calling conventions. Web call declarations use Java-style *modifiers* to specify calling conventions and other technical information. Calling conventions are specified by using either **jsoncall** (default), or **tdacall** modifier. Depending on whether the given web call can access web memory or whether it requires an authenticated user, additional modifiers are introduced:

- Private and project-related web calls (default, no modifiers needed). Such web calls require an authenticated user session as well as an MRAM slot, where the web call can modify data. Such web calls are usually specific to the corresponding web application.
- Private and static web calls (denoted by the **static** modifier). Such web calls require an authenticated user session but do not need an MRAM slot. Such web calls can implement user-specific actions that are independent on a particular web application. Examples of such web calls include `uploadFile` and `set/getUserRegistryValue` [6].
- Public and static web calls (denoted by both **public** and **static** modifiers). Such web calls neither require an authenticated user, nor an MRAM slot. Such web calls can be accessed by any user. An example of a public and static web call is `getAppPropertiesByFullName`, which returns meta-information of the given installed webAppOS app.

A web call may not be public and non-static at the same time.¹⁹

Additional independent modifiers are:

- **inline** - used for fast simple (usually, technical) actions that do not require web processors for execution; inline actions must use the *staticjava* web call adapter to minimize server load; inline calls are executed by the bridge directly, without passing them to web call adapters. The default behavior (without the inline modifier) is to pass the web call to some appropriate web processor.
- **single** - if multiple users are connected to the same MRAM slot, this modifier instructs webAppOS to invoke the given web call only for one user, who originated the parent web call (the caller). The default behavior (without the *single* modifier) for webAppOS is to synchronize web calls between all users connected to the given MRAM slot.

All web calls declarations are stored in *.webcalls* files, which are usually located in the *webAppOS etc* directory or the corresponding web application directory (within the *webAppOS apps* directory). All such *.webcalls* files are automatically parsed and registered at the server side; thus, the corresponding action names (=webAppOS code references) are considered legitime by webAppOS [21].

5. Web Processors

Web processors can be located either at the server side or at the client side. Since the server and the client usually have different environments and technology stacks, it would be difficult to provide an illusion of a single processor and at the same time to use all the benefits each environment offers. Thus, the web computer architecture resembles the multi-processor architecture, where multiple (different) processors share data memory but have separate arithmetic and logic units (ALUs).

Each web processor must be able to execute the code developed for at least one particular instruction set. Usually, a web processor is based on a code interpreter or a set of libraries (e.g., the OS kernel + certain preinstalled libraries). For example, the web processor implementing the “Python 2.7” instruction set can utilize the Python 2.7 interpreter. The precompiled x86_64 Linux kernel 4.x with POSIX libraries can be viewed as a web processor for the “Linux 64” instruction set. Similarly, the web browser can be considered a web processor implementing the “JavaScript 6” and ”WebAssembly 1.0” instruction sets.

By a convention, a web processor implementing some particular instruction set should also offer its more generic variations. However, this is not a rule.

5.1. The Client-Side and Server-Side 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 node, and processing web calls²⁰. When one side makes a web call that has to be executed at the other side, both bridges also communicate on how and where to execute code.

¹⁹If a web call is not static, then it has an MRAM slot. Each slot must be associated with at least one connected user. Thus, an authenticated user session is required, and, hence, the web call is not public.

²⁰The server-side bridge is implemented in Java; it provides a web socket server-side endpoint for clients willing to connect to MRAM slots. The client-side bridge is implemented in JavaScript (located in *webappos.js*); web memory synchronization is performed via a *WebSocket* object.

Each bridge follows the event loop model for processing web calls. The event loop, in essence, is a FIFO queue processed in a single thread.

The client-side bridge relies on the built-in JavaScript even loop. Each web call is enqueued via the *webappos.webcall* function (from *webappos.js*).

The server-side bridge implements its own event loop model called *the queue of web calls*. The component, which implements this queue, is called *Web Caller*.

One MRAM slot can be associated with at most 1 server-side web processor executing code on that MRAM slot. Thus, server-side web calls over the same MRAM slot are executed according to the event loop model. However, since there are multiple parallel MRAM slots, the queue of web calls is actually a double queue. The outer queue is the list of MRAM slots, for which there are non-empty queues of submitted web calls. Each MRAM slot maintains the inner queue of web calls that are waiting to be performed on that MRAM slot.

The server-side bridge takes out MRAM slots from the outer queue in the FIFO manner. If the top-most MRAM slot is already busy (i.e., some web processor is executing code on that slot), the slot is re-enqueued to the end of the queue. Otherwise, the inner queue of web calls waiting for the current MRAM slot is considered: the first waiting web call is being de-queued, some appropriate free web processor is found, and the web call is launched (asynchronously) on that web processor. If there are no free web processors, the web call remains in the queue, and the current MRAM slot is re-enqueued into the outer queue as if the MRAM slot was busy.

5.2. Implementing Client-Side Web Processors

Currently, there is only one built-in client-side web processor implemented in *webappos.js*. The client-side web processor relies on web calls adapters, which can utilize various client-side browser technologies, such as JavaScript (preferred), WebAssembly, ActiveX, Java applets, Flash, etc. JavaScript-based web calls adapters can access web memory directly via JavaScript wrapper objects (from *tda.model*), while non-JavaScript adapters may require additional means to access web memory.

In the future, support for multiple client-side web processors (e.g., via web workers or standalone executables running at the client side) can be added.

5.3. Implementing Server-Side Web Processors

We propose to implement server-side web processors as separate OS processes. Since we use memory-mapped files for implementing web memory, web processors will also be able to access them. However, a web processor (accessing data in web memory intensively) and the model synchronization module (synchronizing changes with the client) may need to access the same web memory concurrently. Thus, the underlying model repository AR implements specific access bits in memory-mapped files to handle concurrent access.

The idea of launching separate OS processes has an important benefit: if the code 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), we can forcefully terminate the web processor and then relaunch it for executing other code. However, upon web processor termination, we have to invalidate the corresponding web memory (this resembles an ordinary application crash when data from RAM are being lost).

Since a single web processor has to implement a certain instruction set such as Java or Python, it reserves some amount of the server RAM for the underlying libraries (e.g.,

Java virtual machine can take several hundreds of megabytes). In addition, the number of web processors that can execute code in parallel is related to the number of physical processor cores. Thus, obviously, the number of web processors running within the same server will be quite small. When there are too many requests for executing code, or when certain web processors require a specific environment, which is incompatible with the current webAppOS installation (e.g., a web processor requiring a different OS), *remote web processors* can be implemented.

5.4. Implementing Remote Web Processors

Remote web processors are like server-side web processors running on additional servers, perhaps, in different environments. The code (actions for web calls) can be shared between all such web processors via a shared or replicated directory. From the webAppOS point of view, a remote processor is just like an ordinary processor, which is accessed by a different adapter (adapter for remote web processors). However, there is an issue on how to share web memory between MRAM located on one node and a remote web processor located on a remote node. There are 3 possible approaches:

- Continue using memory-mapped files. However, they have to be located on a shared network drive. That requires support from the AR repository, so its access bits must take into consideration network delay since memory fragments mapped to files on a network file system are not being synchronized instantly.
- Synchronize web memory in the same way as between the server and the client, i.e., using Web Socket Bus. While we do not need memory access bits, we face another dilemma: we either need to synchronize web memory with a remote processor continuously (thus, wasting network traffic and remote server resources), or to synchronize the whole web memory each time a remote processor is invoked (thus, every call to a remote web processor would become inefficient).
- Use another inter-process communication mechanism (such as Java Remote Method Invocation, RMI) to forward RA-API calls from a remote web processor to MRAM.

Each of these approaches can be used when implementing remote web processors.

5.5. Web Processor Bus Service

To be able to use server-side or remote web processors, the server-side bridge initializes Web Processor/Web Memory Bus as follows. First, the bridge starts an internal service called *Web Processor Bus Service*. This service ensures that web processors can communicate with the bridge. The communication channel is called *Web Processor Bus* [7]. Then, the bridge launches (or connects to) web processors mentioned in the webAppOS *etc/webproctab* file [22]. Each web processor is launched via the corresponding web processor adapter [38]. When initialized (or when it becomes available), the web processor communicates back to Web Processor Bus Service and registers itself there (by calling the *registerWebProcessor* function) [8].

6. Web I/O Devices

The server-side and the client-side code can access virtually any device. For instance, the server-side code can access some database or the file system, while the client-side code can manipulate the HTML document (DOM) and handle UI events. Web I/O devices

implemented via third-party services can also be accessed from code. Thus, a web I/O device can be accessed by making web calls to code that is able to send signals to this device. If a web I/O device itself is a signal source (e.g., it acts as event source such as for user input event within the browser window), a TDA Event object can be created within web memory, and a corresponding TDA event listener will be called by means of a web call.

However, the following two points are required for each web I/O device:

1. The user has to be authenticated to access the device (e.g., a token must be obtained to access a remote cloud drive or users' files stored at webAppOS server).
2. Some device API must be available.

Section 6.1 describes a universal way used by webAppOS to authenticate access to web I/O devices. Section 6.2 lists several predefined webAppOS web I/O devices and their APIs. Section 6.3 shows how third-party file systems and databases can be mounted within the predefined webAppOS web I/O devices.

6.1. Scopes

To be able to access webAppOS predefined web I/O devices as well as external web I/O devices (such as cloud services), authentication is required. 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 (or identifier), which is defined by the *scopes provider* (such as webAppOS or Google) that hosts the resources represented by a scope. For instance, Google defines multiple scopes such as "profile" (<https://www.googleapis.com/auth/userinfo.profile>) and "spreadsheets" (<https://www.googleapis.com/auth/spreadsheets>)²¹.

To be able to access resources within the given scope, authentication is required. For instance, Google's scopes can be accessed after entering the Google account password and by allowing the given web application (such as webAppOS itself or one of its web applications) to access the required scopes (Google login page will inform the user about the requested scopes). Since authorizing scopes requires user's intervention at the client-side, webAppOS API for accessing scopes are also client-side. If some server-side code needs access to some scope, it can make a web call to some client-side code that will authenticate the desired scope.

6.1.1. Scopes API

webAppOS API for accessing scopes is implemented in *webappos.js*. The two main functions are *webappos.request_scopes* and *webappos.sign_out* [9].

The *webappos.request_scopes* function authenticates the user to use one or more particular scopes of the given provider. webAppOS will use the underlying *scopes driver* tailored for the given scopes provider [10]. The driver usually displays a login window, obtains access tokens, and stores them somewhere in the browser memory (e.g., JavaScript variable, localStorage, or cookies) and/or in webAppOS Registry under the "xusers/<login>/<driver_name>" registry branch. Tokens stored in Registry must be persistent offline tokens so that they can be re-used later after the user closes the browser window.

²¹see <https://developers.google.com/identity/protocols/googlescopes> for the full list of scopes

The *webappos.sign_out* function revokes all client-side access tokens from the browser memory. This function, however, does not revoke persistent offline tokens stored in Registry. Those tokens must be revoked separately when the user would desire to revoke them; webAppOS will consult the "xusers/<login>/<driver_name>" Registry branch to revoke the tokens correctly.

6.1.2. webAppOS Scopes Driver

webAppOS itself follows the scopes conventions: it registers the *webappos_scopes* driver and provides the following webAppOS scopes:

- "login" - allows authenticated users to access particular predefined web I/O devices (their home filesystems and the corresponding "users" registry branches); allows the users to make private web calls;
- "project_id" - includes the "login" scope, but also ensures a connection with an MRAM slot for the desired project (project_id has to be either specified in the URL, or selected via the file browse dialog).

Client-side JavaScript example:

```
webappos.request_scopes("webappos_scopes", "project_id");
```

6.2. Predefined Web I/O Devices

6.2.1. webAppOS Registry

webAppOS Registry is a hierarchical database for storing settings related to users, projects, and applications. Like Windows Registry, webAppOS Registry has the following predefined root keys:

- **xusers** - stores restricted information about webAppOS users, such as user's verified e-mail. Data in the xusers key are intended to be accessed only by webAppOS itself; the xusers key is not made accessible for end-users or client-side scripts. Thus, for example, the user cannot change the e-mail without verification by webAppOS.
- **users** - stores information about webAppOS users, where users have full access to their data after they have been authenticated. The users key stores authentication tokens and hashed passwords. The users can change their authentication information (e.g., change a password or add/remove access tokens).
- **projects** - stores information about webAppOS projects. Data in the projects key are intended to be accessed only by webAppOS itself. Since webAppOS projects are usually zipped, some temporary directories are needed to extract them. The projects key stores the mapping between zipped and extracted projects.
- **apps** - stores information about webAppOS web applications and web services. Data in the apps key have read-only permissions when accessed from the client-side. However, installed (and, thus, verified) applications and services running at the server side have read-write access (usually, they write to the branch associated with the app itself; however, they can also access other branches, e.g., to change the configuration of another app).

Each setting in webAppOS Registry can be accessed by a path string, e.g., “xusers/[login]/email”. A path string can reference a final primitive value or a subkey. In the latter case, the value is a JSON object.

Certain webAppOS functionality relies on webAppOS Registry:

- webAppOS login service uses the *xusers* and *users* keys for authenticating existing users and registering new ones (their e-mails are also validated by the service) [11].
- webAppOS MRAM uses the *projects* key to associate MRAM slots with temporary project directories.
- webAppOS server uses the *apps* key to store information about how web services have to be launched (automatically or not).
- webAppOS applications and services can authenticate users for accessing remote web services; user credentials (e.g., access tokens) and other technical information can also be stored in webAppOS Registry, under the *user* key. For instance, webAppOS Home File System searches in Registry for mount points used to access remote storage.

API. webAppOS Registry is accessible via server-side API (accessible via Java object `org.webappos.server.API.registry`) [12]. There are also 2 web calls actions — `webappos.getUserRegistryValue` and `webappos.setUserRegistryValue` — which can be used from server-side and client-side code.

Each user can mount remote registries as subkeys under the corresponding `users/<login>` webAppOS Registry branch. Information about registry mount points is also stored in webAppOS Registry under `users/<login>/registry_mount_points`. Registry API takes care of handling requests to the mounted registry subkeys via the corresponding registry drivers [13].

Scalability. By default, webAppOS Registry is configured to use lightweight JSON files. However, webAppOS Registry can be configured to use the CouchDB no-SQL database. In the latter case, webAppOS Registry can be easily scaled and shared between multiple webAppOS instances (servers), since CouchDB has a built-in replication feature.

6.2.2. webAppOS Home File System

webAppOS Home File System corresponds to the `webappos/home` directory, where each subdirectory corresponds to a home directory of some webAppOS user.

Certain webAppOS services rely on webAppOS Home File System:

- File Browser service - a lightweight HTTP-based API for accessing user’s home file system; used by the File Browser application and `webappos.desktop.browse_for_file` function from `webappos.js`.
- webDAV service - webAppOS service implementing webDAV protocol.

API. webAppOS Home File System is accessible via server-side API (accessible via Java object `org.webappos.server.API.homeFSRoot`) [14]. There are also several file system-related web calls actions [6].

Each user can mount remote file systems as virtual subdirectories of their home directories. Information about such mount points and users’ credentials is stored in webAppOS Registry under `users/<login>/fs_mount_points` [15]. Home File System API takes care of handling requests to the mounted subdirectories via the corresponding remote file system drivers.

Scalability. The webappos/home directory can be shared (or replicated) between multiple webAppOS instances using traditional directory sharing technologies such as Samba or NFS.

6.2.3. E-mail Sender

E-mail Sender is a web I/O device, which is able to send e-mails.

webAppOS login service relies on E-mail sender, when registering new users (to validate their e-mails) and when changing users' passwords.

API. E-mail Sender is accessible via server-side API (accessible via Java object `org.webappos.server.API.emailSender`) [16].

Scalability. Since the e-mail server is specified by its URL and credentials, the same settings can be used in all running webAppOS instances. We assume that mail server load balancing can be configured in the mail server itself.

6.2.4. Desktop

The desktop can be considered a client-side web I/O device implementing a desktop environment with an application launcher, window manager, etc. Particular web applications are running in iframes corresponding to desktop windows. The desktop uses cross-frame messages to ensure the communication between a web application and the desktop. In its simple variant, the desktop can be just a simple tab with the “Sign out” button and the ability to show the list of all installed web applications (as it is done in Microsoft Office Online, iCloud, and Google Docs).

API. While there can be different implementations of the desktop application, they must implement the same Desktop API, which is accessible to webAppOS applications via the `webappos.desktop` object from `webappos.js`. For instance, there are functions `webappos.desktop.launch_app` and `webappos.desktop.browse_for_file` [9].

Scalability. Since Desktop is a client-side web I/O device, each user has their own desktop running on their own browser; thus, there is no need to support Desktop scalability from webAppOS servers.

6.3. Mounting Remote File Systems and Registries

6.3.1. Remote (Cloud) File System Drivers

Scopes drivers can provide access to remote file systems (e.g., cloud drives such as OneDrive or iCloud drive). For that, the scopes driver has to provide one of or both the server-side and browser-side implementation of the File System API (the browser-side API must be implemented only when the driver is intended for serverless mode) [14]. A File System API implementation is called a *file system driver* [15].

Each file system driver is associated with some URI prefix, e.g., “gdrive:”. To mount a remote file system, a registry key “users/<login>/fs_mount_points/<mount-point-path>” must be created; the value of this key is a remote location preceded by the corresponding prefix, e.g., “gdrive:https://drive.google.com/open?id=1234”. webAppOS Home File System module takes into consideration all mount points from Registry: when a user refers to a mount point, webAppOS forwards the request to the underlying driver.

6.3.2. Remote (Cloud) Registry Drivers

Scopes drivers can provide access to hierarchical remote databases that can simulate Registry behavior. The scopes driver has to provide one of or both the server-side and browser-side implementation of the Registry API. Such implementations are called *registry drivers* [13].

Each registry driver is associated with some URI prefix, e.g., “gsheets:”. To mount a remote registry, a registry key “users/<login>/registry_mount_points/<user-registry-subkey>” must be created; the value of this key is a remote location preceded by the corresponding prefix. webAppOS Registry takes into consideration all such Registry keys and will be able to call the corresponding registry driver when the user accesses the mounted registry subkey.

7. webAppOS Applications and Services

From the end user’s as well as from the developer’s point of view, webAppOS is an infrastructure for running web-based applications and services. webAppOS registers the following URL paths and subdomains (in case the domain is specified) for them²²:

- <domain-or-ip>/apps/<short-app-name> and <short-app-name>.<domain> for webAppOS applications;
- <domain-or-ip>/services/<short-service-name> and <short-service-name>_service.<domain> for webAppOS services.

The main difference between applications and services is as follows:

- applications are delivered as visible HTML pages (which can rely on other technologies such as client-side CSS/JavaScript/WebAssembly or server-side PHP);
- services provide some web-based API (HTTP REST/AJAX or non-HTTP), which is invisible to end users but can be used by developers or administrators.

webAppOS applications can rely on webAppOS services. For example, the webAppOS login application requires the login service.

7.1. webAppOS Web Applications

A webAppOS application can be considered as a set of web calls actions. Some of these actions are implemented at the server-side using server-side technologies, server-side web calls adapters, and org.webappos.server.API, while others rely on client-side technologies, client-side web calls adapters and webappos.js.

Depending on how web application HTMLs are delivered, different app adapters can be used. The main adapter is “html” - it just servers the *web-root* directory of the given app (and merges it with the global web root directory *webappos/web-root*). Other adapters such as server-side PHP interpreter or WAR loader can be implemented in the future.

Web applications that require web memory just request the webAppOS “project_id” scope, and webAppOS will ensure the connection to the corresponding MRAM slot. Other applications can suffice without web memory (such as the login application, which is not even able to connect to MRAM, since the user has not been authenticated yet). Still, such applications can rely on public static web calls and certain web services.

²²There can be slight variations in the sub-domain in case the domain is not specified, and the corresponding application or service requires a root path.

7.2. *webAppOS Engines*

Engines resemble shared libraries: their code is shared between different web applications, while data are project-specific and stored within the project web memory. Engines usually implement graphical presentations, e.g., dialog windows or graph diagrams. Engines can provide different implementations depending on how webAppOS will be launched - as a web application, a desktop application, or a mobile application. However, if multiple implementations are provided, each implementation should support the same set of web calls actions. That would allow the code using the engine to be executed on different platforms without modifications.

webAppOS engines resemble TDA graphical presentation engines²³ with the following distinction: in TDA, engines must be implemented using just one adapter (e.g., `staticjava`), while in webAppOS, parts of the same engine can be implemented using different adapters. Thus, it is possible to define an engine, where some commands are implemented at the server-side in Java or Python, while other commands are implemented at the client-side in JavaScript or WebAssembly.

Each engine must provide an interface metamodel, which will be automatically loaded into web memory. Before calling an engine, the application fills the model repository with data according to the metamodel of that engine and submits a command object (links it to the `TDAKernel::Submitter` object). Then webAppOS will search for the web call action with the name equal to the name of the command class, and execute it. Engines, in their turn, can emit events, which must be stored in web memory and also linked to the submitter object. On each event, webAppOS finds the corresponding event listeners and calls them.

7.3. *webAppOS services*

webAppOS services are traditional web services tailored to be installed and executed within the webAppOS environment. There can be different service adapters. The default is “`javaservlet`”, which looks for the given Java class extending `javax.servlet.http.HttpServlet`. Other adapters such as PHP adapter, or an adapter being able to launch a preconfigured virtual machine (or a container) and forward webAppOS ports there, can be implemented in the future.

The following services come out-of-the-box with webAppOS:

- **login.service** - implements user registration and authentication [11]. The user is identified by a password or a token validated at the server-side. OAuth 2.0 authentication via third-party services (e.g., Google or Facebook) can be implemented in the future.
- **webdav.service and FileBrowser.service** - services providing access to the file system via the webDAV protocol and the Dojo REST conventions [17, 18]. The latter service is used by the FileBrowser application. Since both services rely on the File System API, all mounted remote directories are also accessible via these services [14].
- **webcalls.service** - provides access to Web Caller for making web calls [19]. While unauthenticated users can access only public static web calls, authenticated users can also make private web calls such as get the list of their open projects or access

²³Refer to <http://tda.lumii.lv/tda/> for more details on TDA engines.

their *users* subkeys in Registry. If the *project_id* is specified, the authenticated user can also make non-static calls requiring access to web memory.

7.4. Serverfull and Serverless Modes

In serverfull mode (the default mode), both client-side and server-side parts of webAppOS can be used by web applications. In particular, server-side code is able to use webAppOS server-side API (*org.webappos.server.API*), while the client-side code uses *webappos.js* [20, 9].

As a particular use case, webAppOS can be used to develop lightweight serverless applications. “Serverless” is only from webAppOS point view since such applications do not rely on any webAppOS server-side code. However, they can use third-party services such as cloud storage.

Serverless webAppOS applications must meet the following criteria:

- require a simple server (for serving plain HTML/CSS/JavaScript files);
- do not rely on webAppOS server-side API (*org.webappos.server.API*), but can rely on *webappos.js*, which re-implements certain server-side web calls at the client-side;
- do not use *webappos_scopes* driver, but can use other scopes that do not require webAppOS server (e.g., *google_scopes* driver is able to access Google services without the presence of webAppOS server; thus, this driver can be used in serverless mode).

Serverless services, in their turn, have to meet the following requirements:

- They have to be implemented in plain HTML/JavaScript. The desired implementation is to put *index.html* into some subdirectory within the web root directory (e.g., *services/my-service*). This subdirectory will serve as a service request path. The *index.html* file can support some URL query arguments. Thus, a request to the service can look like:

```
<domain-or-ip>/services/my-service/?arg1=value1&arg2=value2
```

The same service can be also implemented in a serverfull variant. If the server-side code (e.g., Java servlet) implements the same path and takes the same arguments, then the service can be equally used from both serverless and serverfull modes.

- If the service connects to some third-party cloud service, some API key (or other credentials) may be required by that third-party. In this case, these credentials have to be accessible by client-side code. That can be implemented by storing them in some JSON file located in the same directory as other files, or hard-coded scripts. However, there is a risk that these credentials can be misused. In contrast, serverfull services can store these credentials at the server-side, e.g., in the *xusers* Registry branch, without direct access from the client.

8. Overall webAppOS Architecture

Figure 4 provides a summary of the webAppOS architecture.

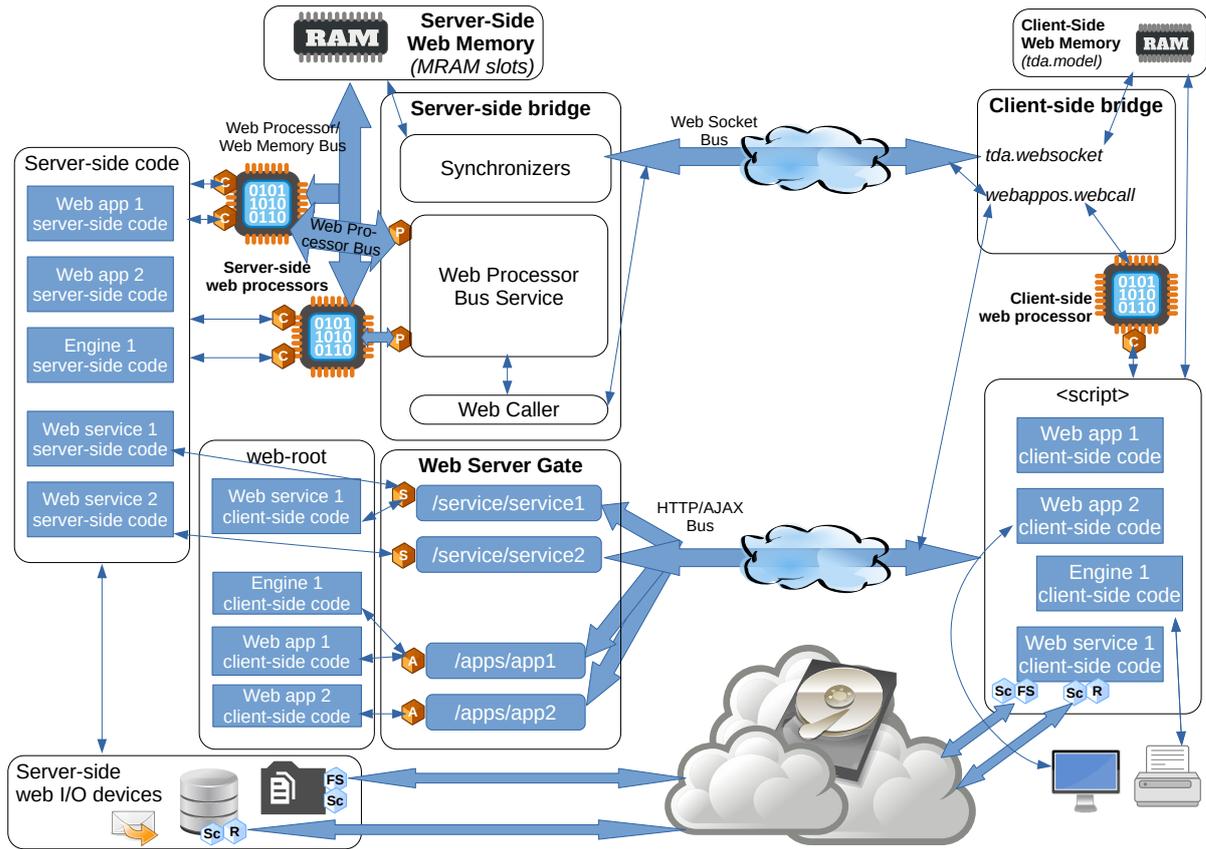


Figure 4: Overall webAppOS architecture.

8.1. Main Components

The main components shown in Figure 4 are:

- server-side web memory containing MRAM slots for all connected users;
- client-side web memory for a particular user working with a particular project; implemented in JavaScript objects accessible via *tda.model*;
- the server-side bridge; it contains web memory synchronizers, Web Caller (for making web calls from server-side code), and runs Web Processor Bus Service (for communicating with server-side web processors);
- the client-side bridge; it contains the *tda.websocket* object (a JavaScript WebSocket instance) for synchronizing web memory; it also provides the client-side *webappos.webcall* function acting as an analog of server-side Web Caller;
- server-side and remote server-side web processors, which are accessed via web processor adapters (“P”);
- the main client-side web processor (currently, there is only one client-side web processor corresponding to the main JavaScript message loop; it does not need an adapter, since the client-side bridge is able to use it directly); in the future, webAppOS might be able to support multiple client-side web processors by means of web workers or browser plugins;

- web calls adapters (“C”) used by web processors as abstraction layers for different instructions sets;
- server-side I/O devices (including Registry, webAppOS Home File System, and E-mail Sender);
- client-side I/O devices (such as the desktop and printers);
- cloud services, which use scopes drivers for authentication (“Sc”); some scopes drivers can provide remote registry drivers (“R”) and/or remote file system drivers (“FS”) for mounting remote registry keys and remote file systems;
- Web Server Gate serving URLs for installed webAppOS applications and services; end users connect to these URLs;
- webAppOS applications (using the code from webAppOS engines) and services; applications and services are attached via various adapters (“A” and “S”) depending on the underlying technology used in that application of service.

Besides main components, there are also several communication channels for transferring data between them. While some channels are simple function calls (depicted as thin arrows in Figure 4), others, which we call *buses*, require sending data between different OS processes or between different network nodes (bold arrows). There are the following main buses in webAppOS:

- Web Processor/Web Memory Bus is used to access MRAM slots from the server-side bridge and server-side web processors; implemented via memory-mapped files; it is hidden behind Web Processor Bus;
- Web Processor Bus is a communication channel between Web Processor Bus Service and server-side web processors; implemented in Java RMI, but can rely also on other technologies (e.g., when accessing remote web processors);
- Web Socket Bus - ensures web memory synchronization and transfers web calls, when they have to be executed on the other node;
- HTTP/AJAX Bus - traditional way for delivering HTML pages and web services; implemented in the underlying webserver software and in the web browser. In case a webAppOS service implements a non-HTTP-based protocol, webAppOS will still use the HTTP/AJAX bus for that service to display relevant information about that service.

An interactive version of Figure 4 with links to the underlying API specifications online is available at <http://webappos.org/dev/idoc/>.

8.2. *Similarity with the Typical Motherboard Layout*

If we re-arrange the discussed elements of the web computer architecture from Figure 1 and add bridges, we come up with Figure 5(a), where we can notice a similarity with the typical motherboard layout (Figure 5(b)).

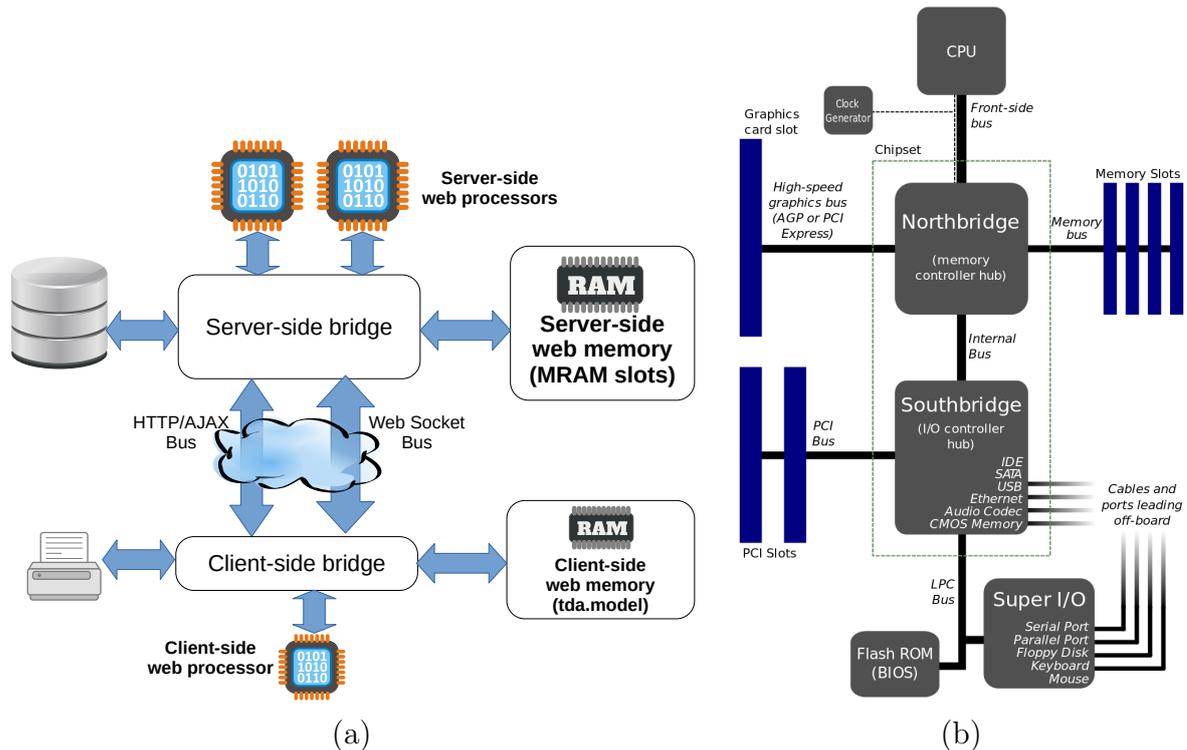


Figure 5: (a) The overall webAppOS architecture. (b) A typical layout of the north and south bridges (image by Gribeco and Moxfyre, CC BY-SA 3.0).

8.3. Developers' Experience

From the developer's point of view, *webAppOS* factors out almost all web-specific aspects; thus, developers can assume they are writing applications for a single PC, with single memory and multiple processors in place. Network communication, user authentication, automatic acquisition of SSL certificates and their renewal, access to user's home directory at the remote file system (where files can be uploaded to), and other features, are provided by *webAppOS* "for free".

8.4. End-Users' Experience

From the end user experience, *webAppOS* resembles Google Drive, Microsoft OneDrive, and Apple's iCloud with the corresponding web applications (text editors, spreadsheets, etc.). There are two modes:

- the desktop mode, which provides classical desktop experience by displaying multiple webAppOS applications in a single browser tab²⁴;
- the full-screen mode, where each webAppOS app occupies a single tab. However, a menu button is provided to launch other apps in new tabs.

8.5. Administrators' Experience and Scaling

The architecture described in this document describes one webAppOS server. For small loads, a single server can be installed by unpacking the binary zip (or by building and installing the webAppOS GitHub project using the gradle tool). The administrator

²⁴The demo can be watched at https://youtu.be/0ycer61_SYw.

has to configure the `webappos.properties` file by specifying the server domain name (or IP), ports, e-mail server address and credentials, optional CAPTCHA server settings, etc. The desired webAppOS apps and services can be installed just by copying them into the `webappos/apps` directory (some of them may require additional configuration). Then, after launched, webAppOS will serve the installed apps and services. If the “secure” setting is enabled in `webappos.properties`, webAppOS will automatically obtain and renew HTTPS certificates via the ACME protocol.

For scaling, some cloud load balancing service (such as Amazon Fargate) can be utilized to manage multiple webAppOS servers. During load balancing, all connections to the same webAppOS project must be forwarded to the same server, since MRAM slots allocated within one webAppOS server are not shared with other webAppOS servers. Each webAppOS instance usually has its own set of web processors (however, they can share remote web processors). Web I/O devices must support scaling (in Section 6, we have discussed scaling of predefined web I/O devices included in the standard webAppOS distribution).

References

Cited API Specifications (Deliverable 2.2)

- [1] “Client-side web memory API,” <http://webappos.org/dev/idoc/?=cswmapi>.
- [2] “Web processor/web memory bus,” <http://webappos.org/dev/idoc/?=wpwmb>.
- [3] “Web socket bus,” <http://webappos.org/dev/idoc/?=wsb>.
- [4] “Server-side web calls adapters,” <http://webappos.org/dev/idoc/?=sswca>.
- [5] “Client-side web calls adapters,” <http://webappos.org/dev/idoc/?=cswca>.
- [6] “webappos.* web calls actions,” <http://webappos.org/dev/idoc/?=wwca>.
- [7] “Web processor bus,” <http://webappos.org/dev/idoc/?=wpb>.
- [8] “Web processor bus service,” <http://webappos.org/dev/idoc/?=wpbs>.
- [9] “webappos.js API,” <http://webappos.org/dev/idoc/?=wjsapi>.
- [10] “Scopes drivers,” <http://webappos.org/dev/idoc/?=sd>.
- [11] “webAppOS ”login” service,” <http://webappos.org/dev/idoc/?=wls>.
- [12] “Registry API,” <http://webappos.org/dev/idoc/?=rapi>.
- [13] “Registry drivers,” <http://webappos.org/dev/idoc/?=rd>.
- [14] “File system API,” <http://webappos.org/dev/idoc/?=fsapi>.
- [15] “File system drivers,” <http://webappos.org/dev/idoc/?=fsd>.
- [16] “E-mail sender API,” <http://webappos.org/dev/idoc/?=esapi>.

- [17] “webDAV service,” <http://webappos.org/dev/idoc/?=wds>.
- [18] “FileBrowser service,” <http://webappos.org/dev/idoc/?=fbs>.
- [19] “Web calls service,” <http://webappos.org/dev/idoc/?=wcs>.
- [20] “webAppOS server-side API,” <http://webappos.org/dev/idoc/?=wssapi>.

Cited webAppOS Documentation (Deliverable 3.2)

- [21] “.webcalls file format,” <http://webappos.org/dev/idoc/?=wcff>.
- [22] “webproctab file format,” <http://webappos.org/dev/idoc/?=wptff>.

Other References

- [23] J. von Neumann, “First draft of a report on the EDVAC,” Moore School of Electrical Engineering, University of Pennsylvania, 1945.
- [24] V. Bush, “As we may think,” *SIGPC Note.*, vol. 1, no. 4, pp. 36–44, Apr. 1979.
- [25] Ecma International, “ECMA-262 6th edition, the ECMAScript 2015 language specification,” 2015.
- [26] A. Chiesa, E. Tromer, and M. Virza, “Cluster computing in zero knowledge,” in *Advances in Cryptology - EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 371–403.
- [27] Object Management Group, *OMG Meta Object Facility (MOF) Core Specification Version 2.4.1*, Object Management Group Std. formal/2011-08-07, 2011.
- [28] I. Kurtev, J. Bézivin, and M. Aksit, “Technological spaces: An initial appraisal,” in *CoopIS, DOA 2002 Federated Conferences, Industrial track*, 2002.
- [29] J. Bézivin and I. Kurtev, “Model-based technology integration with the technical space concept,” in *Proceedings of the Metainformatics Symposium*, 2005.
- [30] S. Kozlovics, E. Rencis, S. Rikacovs, and K. Cerans, “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*, ser. Frontiers in Artificial Intelligence and Applications, vol. 224. Amsterdam, The Netherlands: IOS Press, 2011, pp. 80–93.
- [31] J. Barzdins, S. Kozlovics, and E. Rencis, “The Transformation-Driven Architecture,” in *Proceedings of DSM'08 Workshop of OOPSLA 2008*, Nashville, Tennessee, USA, 2008, pp. 60–63.
- [32] S. Kozlovics and J. Barzdins, “The Transformation-Driven Architecture for interactive systems,” *Automatic Control and Computer Sciences*, vol. 47, no. 1/2013, pp. 28–37, 2013, Allerton Press, Inc.
- [33] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework, 2nd Edition*, E. Gamma, L. Nackman, and J. Wiegand, Eds. Addison-Wesley, 2008.

- [34] S. Kozlovics, “The orchestra of multiple model repositories,” in *SOFSEM 2013: Theory and Practice of Computer Science*, ser. Lecture Notes in Computer Science, vol. 7741. Springer Berlin Heidelberg, 2013, pp. 503–514.
- [35] R. Liepiņš, “Library for model querying: IQuery,” in *Proceedings of the 12th Workshop on OCL and Textual Modelling*, ser. OCL '12. New York, NY, USA: ACM, 2012, pp. 31–36.
- [36] S. Kozlovičs, “Efficient model repository for web applications,” in *Proceedings of the 13th International Baltic Conference on Databases and Information Systems (Baltic DB&IS 2018)*, ser. CCIS, vol. 838. Springer Nature Switzerland, 2018.
- [37] —, “Fast model repository as memory for web applications,” in *Databases and Information Systems X*, 2019, vol. 315, pp. 176–191.
- [38] “Server-side web processor adapters,” <http://webappos.org/dev/idoc/?=sswpa>.