The Liquid WebWorker API for Horizontal Offloading of Stateless Computations

Andrea Gallidabino and Cesare Pautasso

Software Institute, Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland
E-mail: andrea.gallidabino@usi.ch; c.pautasso@ieee.org;
http://liquid.inf.usi.ch

Received October 2018;
Accepted February 2019

Abstract

As most users access the Web from multiple devices with different characteristics, ranging from powerful desktops or laptops to tablets, mobile phones or watches and cars, liquid Web applications seamlessly flow across multiple Web-enabled devices and adapt their distributed user interface to the set of devices simultaneously accessing the application. In this paper we focus on the business logic layer of rich Web applications and explore the opportunity to reduce the execution time of CPU-intensive tasks or limit their energy consumption by offloading them among nearby devices running the same liquid Web application. We extend the standard HTML5 WebWorker API with the concept of liquid WebWorkers, so that developers can transparently offload parallel execution of stateless tasks by managing the necessary device selection and direct peer-to-peer data transfer. By introducing the liquid WebWorker API into our Liquid.js framework, we present how to create a pool of devices sharing their CPU processing capabilities according to different policies.

Journal of Web Engineering, Vol. 17_6&7, 405–448.
doi: 10.13052/jwe1540-9589.17672

Keywords: WebWorkers, Edge Computing, Liquid Software, Horizontal Offloading.

1 Introduction

As more and more users connected to the Web own more than three devices, ranging from powerful desktops and laptops to smaller devices, such as tablets, smart phones or smart watches [17], this affects the way users interact with their software applications running across their devices [16]. Liquid software [19, 38] empowers users to run their applications across a set of multiple heterogeneous devices. The devices can be used one at a time (when the liquid software flows between them) or simultaneously (when the liquid software flows to fill them all up). In this paper we focus on the second, simultaneous screening usage scenario, where one or multiple users run applications across multiple devices at the same time. In our previous work, we studied how the architecture of a liquid Web application may be designed to take advantage of all available devices [15]. Users interacting with multiple devices may trigger data synchronization activities that will ensure a consistent view over the state of the distributed Web application. Having multiple, partially idle devices also opens up the opportunity to exploit their computational resources to speed up CPU-intensive tasks. In this paper we focus on the business logic layer of the application and show how to transparently offload the execution of CPU-intensive tasks among the active devices on which the application has been deployed. As opposed to vertical offloading which takes advantage of remote Cloud resources [25], in this paper we introduce an horizontal offloading approach, where only local devices are involved.

This paper makes the following contributions:

1. It presents the liquid WebWorker API, a novel abstraction built on top of the standard HTML5 WebWorker API,[1] which allows developers to add parallelism to their liquid Web applications by offloading computational tasks from a device to another;
2. It discusses how the offloading decision can transparently follow different policies, which may take into account user constraints and

[1]https://html.spec.whatwg.org/multipage/workers.html

different trade-offs, e.g., minimizing energy consumption, reducing communication costs or minimizing the task execution time;
3. We describe in detail how the liquid WebWorker concept has been implemented as an extension of the Liquid.js for Polymer framework;
4. We present empirical results showing that the performance of the Web application can be improved by offloading CPU-intensive tasks across different classes of devices.

This article extends our previously published ICWE2018 paper [13] with new sections describing:

1. Fault tolerance aspects;
2. Asynchronous data exchange via liquid properties;
3. Ranking the processing capacity of heterogeneous devices with micro-benchmarks;
4. Usage scenarios of liquid WebWorkers for horizontal computational offloading;
5. New offloading policy addressing security issues.
6. Extended discussion on offloading policy rules taking into account micro-benchmarks and asynchronous data synchronization.

The rest of this paper is structured as follows: in Section 2 we present related work which inspired this paper; in Section 3 we discuss the design of a liquid WebWorker, its internal architecture and then introduce the API it exposes; in Section 4 we discuss some advanced performance-enhancing features we built on top of the liquid Web-Worker core design; in Section 5 we describe in which application scenarios liquid WebWorkers could be deployed and describe some use case scenarios; in Section 6 we present the empirical measurements we observed during the experiments we ran on our prototype; in Section 7 and in Section 8 we summarize the results discussed on this paper and point out additional open research challenges to further improve the design and implementation of liquid WebWorkers.

2 Related Work

The technological foundation of liquid software emerges from the Internet of Things [2], the Web of Things [18], or more in general from the Programmable World [37]. Pervasive Computing [33] shows how microprocessors can be embedded in all sorts of objects scattered around us. Today those “things” are not isolated from each other anymore, they became “smart”, and they are able to communicate with any similar object around them. The users and the devices surrounding them make up a complex ecosystem [41] which requires software to adapt to the set of available devices, for example whenever a smart object enters or leaves the proximity of the user running a given software application. Similarly, liquid software automatically flows between devices to adapt its deployment configuration to take full advantage of the resources and capabilities of multiple devices. Nowadays smart objects and devices are so common [42] and advanced [6], that users may also interact with some devices that they do not directly own, but nevertheless they are allowed to share some information with it in order to run applications across particular devices. For instance, it is possible to find public displays [5] owned by cities [44] which may allow users to interact with them directly or by pairing their mobile smart phones with them. This way, users could for example take advantage of the large screen to display a picture slide show. This is a relevant scenario for liquid software, as the application should run across multiple devices to achieve the user’s goal. More in general, the challenge we focus on in this paper is how to enable devices to share their available computing resources and how to design software which can seamlessly access them.

Mobile Computing [10] discusses the potential of creating powerful distributed systems made of mobile hardware that communicates through the Internet. A mobile computing system trades portability and social interactivity with many distributed systems challenges such as dealing with device connectivity, discovery, trust establishment and proximity detection. The study of context-aware systems [35] allows us to understand how to create systems based on proximity-awereness [7].

Web technologies have been evolving towards increased support for reliable mobile decentralized and distributed systems [12]. In the past decade an effort has been made to improve and create new HTML5 standards [43] that can help with the creation of complex mobile distributed systems able to reliably maintain data synchronized between devices [30]. Thanks to novel standard protocols it is also possible to interconnect any device by using any standard compliant Web browser. Okamoto *et al.* [29] show how to create mobile Web distributed systems by exploiting the WebWorker HTML5 standard.

Web browsers allow any device to connect to a Web application, meaning that users can connect all the devices they own to run a single liquid cross-device application. Whenever a device is connected to the liquid application, the resources it provides are exploited by the software. This approach is similar to the one of Volunteer Computing [1], where users willingly connect their own devices to perform a global computation, and share data, storage or computing resources among them.

Edge computing [36] focuses on optimizing data processing and storage by shifting computations closer to the source of the data, as opposed to shipping a copy of the data to large, centralized Cloud data centers [32]. The optimization reduces bandwidth consumption and latency in the communication between the edge devices, making it possible to reduce the overall processing time of an operation. Fog computing [3, 27] takes edge computing to the extreme, by making it possible to make all data processing computation within the IoT ecosystem. Liquid software also incorporates such performance goals, in order to seamlessly migrate applications among multiple user-owned devices without relying on centralized Web servers. Similar concepts can also be found in the ubiquitous computing [31] literature.

Traditionally large amounts of distributed computational resources was found mainly within clusters of computers or Cloud data centers, however recent trends show that also the Web, by employing Web browsers running across many types of devices and WebWorkers as a programmatic abstraction for parallel computations, can deliver a decentralized computation platform [8] as well. While most existing computational offloading work focuses on shifting workloads *vertically* from mobile devices to the Cloud [9], in this paper we study how to do so *horizontally* by using nearby devices. While these may not be as powerful as a Cloud data center, they will remain under the full control of their owners and enjoy a better proximity on the network.

Hirsch *et al.* [20] propose a technique for scheduling computation offloading in grids composed by mobile devices. The scheduling logic of the system is able to offload a set of heterogeneous jobs to any mobile device after an initial centralized decision-making phase. This is followed by the job stealing phase, in which jobs are relocated to other devices in a decentralized manner. The scheduler considers the battery status, the CPU performance and the uptime expectation of all connected devices when it has to decide where to offload jobs. The CPU performance is computed using a benchmark. While this approach shows promising results and it is able to increase the overall performance of computational-intensive applications, in this paper we present a fully decentralized approach able to operate inside a Web browser, where complete information about the devices hardware and software configuration is not always accessible.

Loke *et al.* [26], propose a similar system allowing multi-layered job stealing techniques also with a hybrid approach (both centralized and decentralized) for offloading decisions. The decision depends on which devices are close to the device that starts the computation and then tries to scatter the job between them. Their approach is not based on a Web browser and relies on Bluetooth or WiFi for inter-device communication.

3 Liquid WebWorkers

Like standard HTML5 WebWorkers, also liquid WebWorkers (LWW) are designed to perform background computations in a parallel thread of execution. Unlike standard HTML5 WebWorkers, the work can potentially be transparently offloaded across different devices. To do so, LWW use a simpler stateless programming model, which helps developers identify the boundaries of the task to be offloaded. Liquid WebWorkers receive discrete atomic jobs to be processed and produce the corresponding results all at once. The computational offloading is kept completely transparent from the developer, who can use specific task placement policies to prioritize the available devices according to different criteria.

3.1 APIs

Liquid WebWorkers take care of executing tasks by invoking the corresponding HTML5 WebWorker. Liquid WebWorkers are organized into a *pool*, whose goal is to manage their lifecycle, transparently choose on which machine tasks should be executed, and reliably dispatch tasks towards the corresponding WebWorker, which can be located either locally or remotely.

The liquid WebWorker pool and the liquid WebWorker expose their own API that can be used by the developer for building multi-device liquid applications. Operations inside the LWW pool are executed asynchronously because they require to communicate with remote devices or exchange messages between the global JavaScript context and the worker. For this reason we decided to deal with asynchronous operations with Promises,[2] which may invoke either a successful or a failing callback upon completion.

A rejected promise may return two types of error: either a *communication error* or an *execution error*. In the first case a failure happens during the offloading of a task from a device to another due to a problem in the sending process, either because there is no connection linking the two devices, because the remote machine is currently unavailable, or because a timeout happened. The second error type is thrown whenever there is a problem with a LWW instance, either because the remote LWW is not yet instantiated or there was an internal error in the LWW execution.

3.1.1 Liquid WebWorker Pool API

Table 1 lists all methods exposed by the liquid WebWorker pool API. The LWW pool can be instantiated by passing the reference to a *sendMessage* function whose signature must accept two parameters: *deviceID* and *message*. This function will be called every time the LWW pool has to deliver a message to another device, it does not matter to the pool how the payload is delivered, but the pool expects

[2]https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Promise

Table 1 Liquid WebWorker pool API [13]

Liquid WebWorker pool API

Constructor
LiquidWebWorkerPool(sendMessageFunction)

sendMessageFunction signature
sendMessage(deviceID, message)

Method name and parameters	**Return value**
createWorker(workerName, ScriptURI)	Promise(workerInstance)
getWorkerList()	Promise(workerNameList)
updatePairedDevice(deviceID, data)	Promise(deviceID)
removePairedDevice(deviceID)	Promise(deviceID)
callWorker(workerName, message)	Promise(response)
_callWorker(workerName, message)	Promise(response)
forwardMessage(message)	Promise()
terminateWorker(workerName)	Promise(workerName)

that the function reliably delivers the whole *message* object to the device labeled as *deviceID*.

The LWW pool API exposes eight methods:

- **createWorker:** instantiates a new LWW and binds it to the LWW pool automatically. The pool may contain any number of workers, limited only by the Web browser configuration and available resources. WorkerNames are unique, if the pool is requested to create a worker with an already existing name, then it will fail and return a rejected Promise. The script can be either a *URI* pointing to a Web resource, or it can be a String containing the actual script code. Both parameters are required.
- **getWorkerList:** this method returns a dictionary object containing all the references to the instantiated LWWs contained in the pool, indexed by the corresponding workerNames.
- **updatePairedDevice:** this method updates the information about the paired devices stored inside the pool. The deviceID is the same that will be passed in the *sendMessage* function whenever it will be called. The data is stored in an object that contains information about all devices. Depending on the policy rules employed, this object may contain different information (see Section 4.4).

- **removePairedDevice:** this method removes a paired device from the stored list of paired devices. The framework will take care of ensuring that any task currently offloaded on the device to be removed will eventually complete. No new tasks will be assigned to the removed device.
- **callWorker:** the function is used to submit a task into the pool, which will be executed either locally or remotely. Once submitted, the pool decides where the task will be executed, then it creates the corresponding promises and calls the sendMessage function if the task is executed remotely, otherwise it will call the _callWorker function.
- **_callWorker:** this method is used to submit a task into the pool for local execution. This method directly pushes the task message into the correspondig local LWW instance and waits for its asynchronous response by setting up a promise object.
- **forwardMessage:** whenever a device receives a message sent from another device using the *sendMessage* function, it must forward such message to be processed inside the pool by calling the *forwardMessage* function.
- **terminateWorker:** this method ends the lifecycle of a LWW instantiated inside the pool. If the workerName is invalid or undefined, it returns an error.

3.1.2 Liquid WebWorker API

Table 2 lists all methods exposed by the LWW API. If an invalid or undefined LWW pool is passed as a parameter of the constructor, then the methods *callWorker* and *_callWorker* will behave equivalently and the liquid WebWorker will never attempt to offload the execution on remote devices. That is because, without being connected with a pool, the LWW cannot determine where the submitted tasks should be executed. The LWW does not store information about paired devices nor it knows if it is paired to other LWWs as this information is managed by the associated pool.

Developers can call methods on the worker instances without the need to proxy their execution requests on the pool, since the liquid WebWorker object itself exposes an API. The LWW exposes three methods:

Table 2 Liquid WebWorker API [13]

Liquid WebWorker API	
Constructor LiquidWebWorker(LWWpool, workerName, scriptURI)	
Method name and parameters	**Return value**
callWorker(message)	Promise(response)
_callWorker(message)	Promise(response)
terminate()	Promise(workerName)

- **callWorker:** this method submits a task into the LWW, if the worker is bound to a LWW pool then it will request the pool if the task should be executed remotely or not, otherwise it will automatically call the *_callWorker* method.
- **_callWorker:** this method bypasses the LWW pool policies and executes the tasks directly on the issued worker locally.
- **terminate:** this function will terminate the WebWorker instantiated in the background, making it possible to safely delete all references pointing to the LWW instance. The termination is immediate and does not wait for the end of the task execution.

3.2 Design

Figure 1 shows the main components of the liquid WebWorker pool running across two devices. Tasks can be submitted from either devices and the pool will decide whether they will be executed using workers of the local pool, or they will be offloaded to other devices.

In addition to the set of workers, the LWW pool stores references to the submitted and the currently executing tasks in the form of *pending promises*. It also maintains information about the *paired devices*:

- **pending promises:** for all submitted tasks, the pool creates a promise that waits for the worker to complete the computation and returns the results by posting the response and its associated unique identifier. The promise contains the callback that must be fulfilled or rejected when the remote device or the local worker responds. The payload of the response contains the identifier of the corresponding promise, which can be easily retrieved from the corresponding dictionary inside the LWW pool.

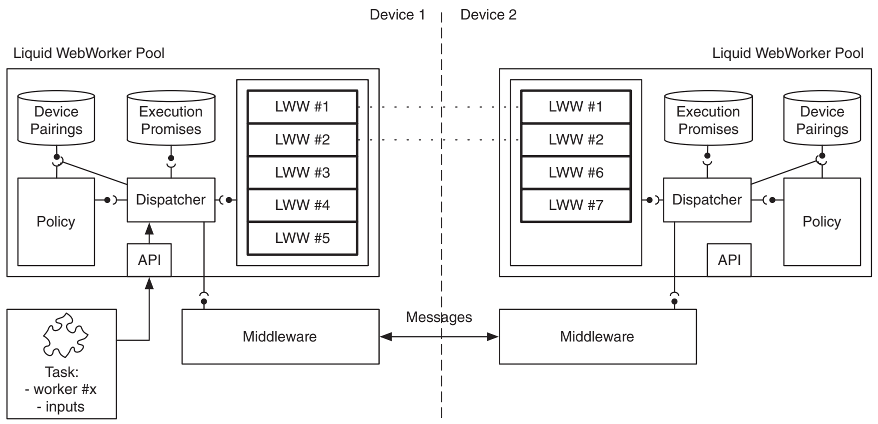


Figure 1 The liquid WebWorkers Architecture. Arrows show the flow of a task and the exchange of messages between clients. Dotted lines indicate *paired* relationships between liquid WebWorker instances [13].

- **paired devices:** the pool keeps track of all paired devices. This information contains the hardware specification of the devices, such as its type (e.g. Desktop or Phone) or any other information useful to the policy component for taking task offloading decisions (e.g. processor specs, battery level, OS version).

The *dispatcher* component forwards tasks to the *right* LWW and thus the *right* device. The decision on where the execution of the task will happen is controlled by the *policy* component, which uses data fed from the *device pairings* storage in order to take a decision. Whenever the dispatcher forwards a task, then it also saves the corresponding callback promise. In the case of remote execution offloading, the dispatcher does not send the task directly to the remote device, but it sends messages through a pre-configured connection middleware (see Section 3.1.1). Each message contains in its payload the corresponding *promise identifier*, the *inputs* of the task that need to be executed, and the *name* of the worker that must be invoked on the remote machine.

The dispatcher component can create new WebWorkers either by passing an URI pointing to a script stored in a central server, or by passing the content of the script as a String that can be directly shared between devices without the need to fetch it from a Web server. In the latter case the dispatcher is able to instantiate the WebWorker script by converting the String to a Blob[3] Object.

LWW are designed to be used for **stateless** computation; in fact, paired workers do not share or synchronize any data among each other. Likewise, every job is treated as an independent computation. Nevertheless it is possible to simulate stateful computations by submitting a task that would include as input the previous state of the worker, and then return the new state with the result so that it can be stored and passed along with the next task. This way, each task of the sequence can still be transparently sent to different devices.

The sequence diagram in Figure 2 illustrates the LWW call lifecycle and how the components inside the LWW pool communicate during local and remote execution. The assumption is that *device1* and *device2* have been paired and workers *w1* and *w2* have been created on both devices. A task addressed to *w2* is submitted by invoking the method *callWorker*. The pool will determine where the task will be executed by invoking the internal *_where* function of the *policy* component. In the first case the *policy* component chooses to execute the task locally. This results in the local call to the corresponding LWW. The *response* is asynchronously computed within the worker and passed as a parameter in the fulfilled promise. Internally, workers use the standard HTML5 postMessage/onMessage API to exchange their input and output data with the LWW pool. This way, from the perspective of the caller, executing a task locally or remotely is indistinguishable.

In the diagram the caller again invokes the *callWorker* method and eventually receives a response inside the fulfilled promise, however inside the pool the process changes whenever the *policy* component chooses to execute the task remotely. In this case the pool first sends a message to the remote device, the remote pool executes the task on a remote LWW and eventually it will send back a response. If no response is received within a given developer-configurable timeout, the LWW pool will attempt to find another device and resubmit the task. If eventually no more remote devices can be found, the task will be executed locally.

[3]https://developer.mozilla.org/it/docs/Web/API/Blob

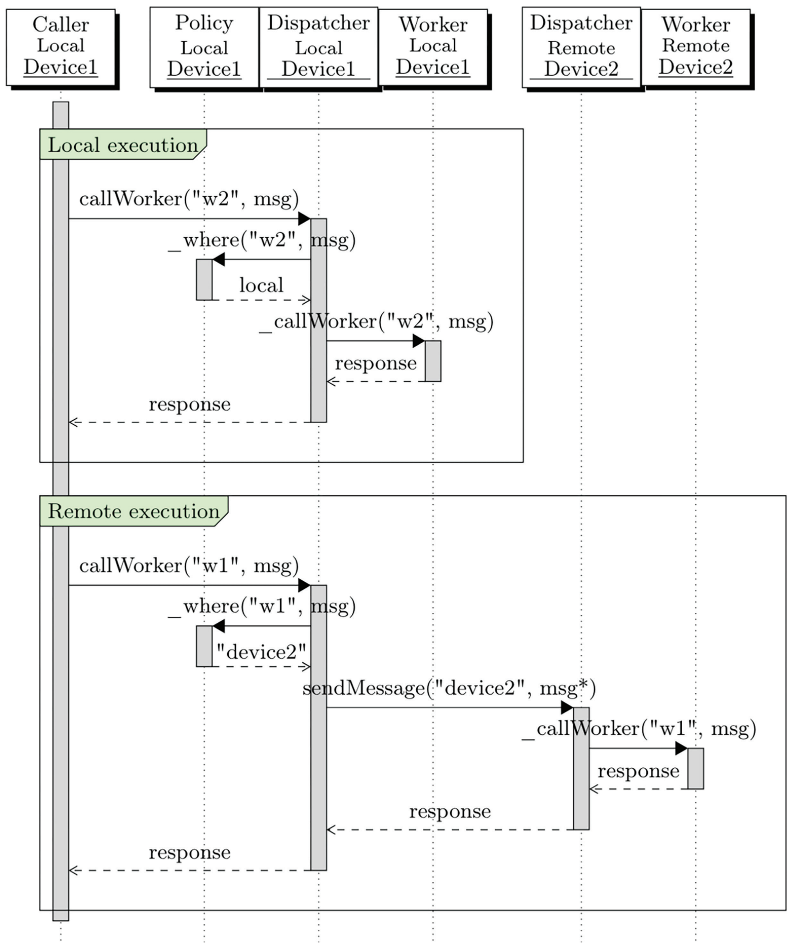


Figure 2 Local and remote execution sequence diagram [13].

3.3 Liquid.js Prototype

We built a liquid WebWorkers prototype within the Liquid.js for Polymer [11] framework. Liquid.js is a Web framework for building decentralized, component-based, liquid Web applications that can be deployed across multiple heterogeneous devices. Applications developed with Liquid.js are built using the Web Components standard, which provides the necessary abstractions to structure the application user interface and its state into units that can be independently deployed across multiple devices.

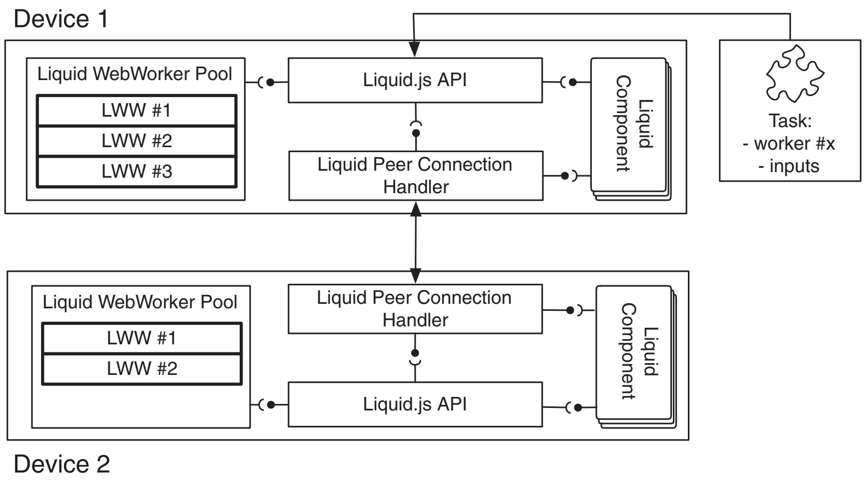


Figure 3 Component view of the implementation of liquid WebWorkers inside the Liquid.js for Polymer framework [13].

Figure 3 illustrates a simplified component view of Liquid.js extended with the LWW pool. The liquid WebWorker pool is managed by the framework itself, hidden behind its own API [14]. The framework manages inter-device communication through a separated component called *Liquid Peer Connection*, which automatically manages and sends messages through peer-to-peer connections using the WebRTC protocol. Developers who wish to use the LWW computation offload feature need to invoke the *callWorker* method exposed by the Liquid.js API. The Liquid.js framework also allows to automatically create workers on other machines whenever the *updatePairedDevice* method is called, which guarantees that a copy of each LWW can be found on all paired devices.

4 Advanced Features

The decision on where a task should be offloaded to, should be taken based on different criteria and following constraints established by the liquid Web application developers, users or device owner preferences. To do so, in this section we outline a number of advanced features that allow to enhance the flexibility and customizability of our LWW prototype.

4.1 Micro-Benchmark

In order to be able to implement valid policy rules inside the LWW pool, we need to predict what are the capabilities of each connected device. Running a macro-benchmark [22] on all the devices before they are allowed to join the liquid Web application would not rank the machines correctly. Macro-benchmarks test the performance of a whole system, however liquid applications are sandboxed inside the Web Browser, which does not give full access to the device resources. For this reason we aim to assess only the resources provided to the Web browser tab that is running the application. Moreover a macro-benchmark is an invasive process meant to be ran stand-alone to avoid interference from other non-idle processes. This would prevent users from interacting with their devices while the benchmark is running, which may take a long time to complete.

In our scenarios we need to be able to predict the capabilities of a device for as long as it connects to the liquid Web application. The amount of available resources provided by the device may dynamically change at runtime, because users can close or open new tabs while they are browsing the application. The benchmark should be repeated over time to accurately track the amount of available resources.

We decided to follow a micro-benchmarking approach [24], which is suited for mobile Web-enabled devices and allows us to test the performance of the active Web browser tab from within the browser itself by exploiting HTML5 standard APIs.

The liquid WebWorker pool runs a micro-benchmark on startup after the pool is instantiated, then it keeps re-running the test at regular intervals. The interval time span is configurable by the developer of the liquid application. The benchmark runs in a background WebWorker and does not prevent the user from interacting with the application while it is executing. The benchmark environment is created with the library *Benchmark.js* [4] and the benchmark testbed can be customized by the developer of the application.

The result of the benchmark represents the average number of iteration per cycle it was able to perform during the execution. We use this number to rank our devices, making it possible to compare their performance.

4.2 Failure Handling

During the task offloading process, failures may happen. The most common failure in distributed systems derives from disconnection of the peers [34], however failures may be generated also by faulty operations such as task executions or faulty policy rules predictions [23].

In Figure 4 we show the expected sequence diagram of the offloading process for liquid WebWorkers with synchronous data transfer:

1. *Device* offloads the *task* to *Device2*;
2. *Devices2* receives the *task*;
3. *Devices2* executes the *task*;
4. *Devices2* submits the response to *Device*;
5. *Device* receives the response and make use of it.

We recognize that in this offloading process failures can happen for three reasons:

- Failed connectivity – the communication between the two devices is interrupted during the offloading process;
- Liquid WebWorker failure – a run-time error during the execution of the offloaded task occurs;
- Timeout – the task does not complete within a given amount of time and the device does not send back a response.

In order to create a reliable system, in this section we propose a solution for all three scenarios.

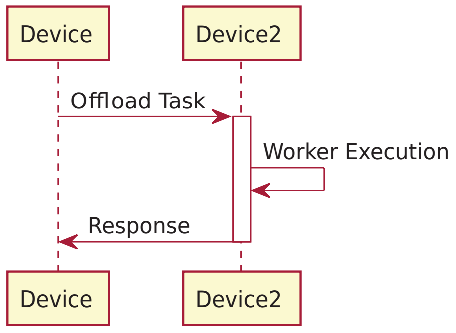
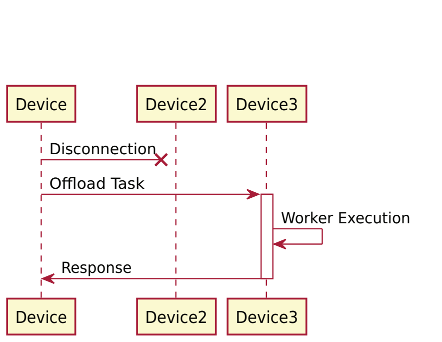
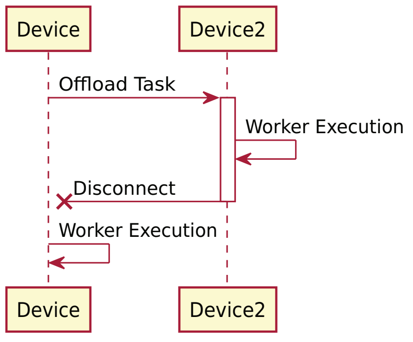


Figure 4 Task offloading without failures.



(a) Disconnection before task offloading



(b) Disconnection after task offloading

Figure 5 Disconnection.

A peer can disconnect anytime throughout the whole task offloading process, Figure 5 shows how to recover the process when a peer disconnects before the task has been completely sent to another device (see Figure 5(a)), and how it recovers when the task has already been offloaded before the peer disconnects (see Figure 5(b)). In the first case (Figure 5(a)) *Device2* can disconnect before the starting device has chosen where to offload the task, after it chose where to offload or even during the task offloading communication. Since the task execution has not yet been started, the disconnected device will be excluded from the ranking of candidate devices and the second most-suitable device will replace it as the new offloading target. In the second case (Figure 5(b)) it does not matter if *Device2* disconnects before, during, or after the *Worker Execution*, in all three cases the starting device will immediately notice the disconnection and by default it will try to recover the execution by running the worker locally. The LWW pool can be configured to retry the execution on the next most-suitable device. If there are no other devices connected, the device will attempt to run the execution locally.

Figure 6 shows how the LWW pool recovers when an error happens during the offloaded task execution, which may crash the LWW. The LWW pool is able to detect when a worker throws an error, catching it immediately and by responding to the starting device about the failed task execution. Such error response also includes the reason

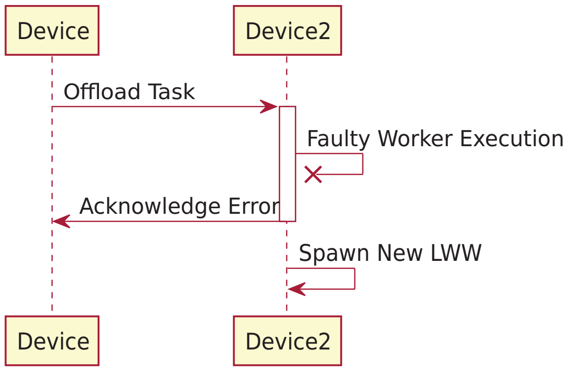
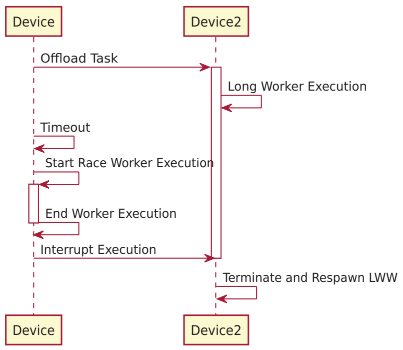


Figure 6 Run-time error during the offloaded task execution. The remote LWW is independently respawn but the local device should decide how to recover the task.

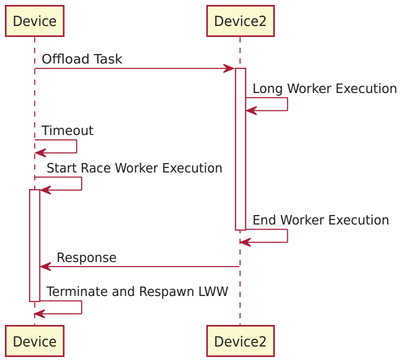
of the crash and will not stop the execution of the liquid application on *Device2*. Once the response has been sent, a new LWW will be spawned on *Device2*, which will return to be available to service other task offloading requests. By default the starting device does not try to locally re-execute such failed task or to offload it to a different device. The decision is left to the developer of the application that must define which recovery operation to execute by handling the error acknowledgement event from the LWW pool.

In the last scenario no task execution response is sent by *Device2* back to the starting device within a given amount of time even if there are no problems with the connection (see Figure 7). This can happen because the LWW pool decided to offload the task to a slow device, or because *Device2* cannot complete the task execution before a timeout occurs. Whenever the timeout triggers, the starting device starts to execute the task locally. This creates a race between the local and remote task execution: if the local task ends before *Device2* has responded, then the starting device notifies *Device2* that it does not need its answer anymore, when *Device2* receives the message it will terminate and respawn the corresponging LWW. If *Device2* answers before the starting device finishes, then the opposite happens and the start device terminates and respawns the LWW.

Developers can set the default timeout as part of the LWW configuration and also associate a different timeout with each task. If the timeout



(a) The starting device wins the race



(b) *Device2* wins the race despite the timeout

Figure 7 No disconnection with timeout and local task re-execution race.

is set to *zero*, then the LWW pool will always start a race between the local device and the remote peer. In this case the LWW will attempt to compute tasks with the highest speed among two devices, however it will also increase the energy consumption on both devices.

4.3 Synchronous vs Asynchronous Data Transfer

In Section 3.2 we showed that the messages exchanged between devices also contain the corresponding input data that has to be passed to the LWW in order to complete the task. In our prototype we deploy the LWW pool on top of the Liquid.js framework, which already transparently and automatically synchronize the state of liquid components shared between devices [11].

If the data used inside the LWW pool is stored in a *liquid property*, then we do not have to send it with the task offloading message, because it is already synchronized between the devices. In Figure 8 we show that we can abstract and separate the flow of data and task offloading with two different channels. Data is synchronized between all paired liquid components, while tasks offloading messages are exchanged between the LWW pools. Whenever the data should be loaded or saved in liquid property, then the LWW pool is allowed to interact with the liquid components directly. To take advantage of this feature, the developer of the application must call the LWW from inside the liquid component,

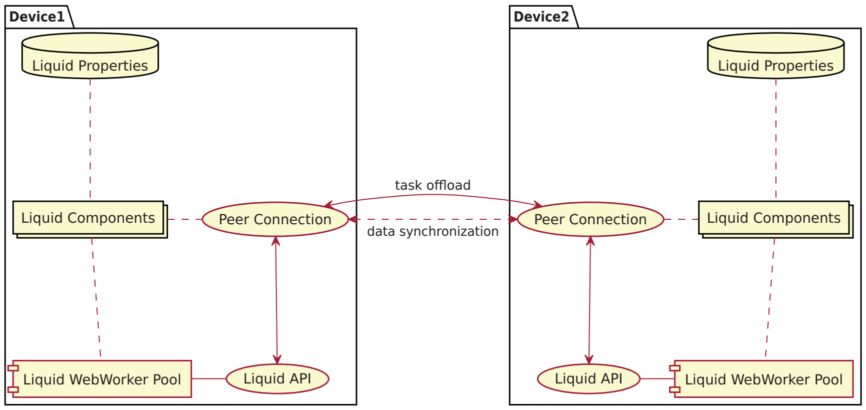


Figure 8 Asynchronous data transfer: the dashed lines represent the flow of the data, the full lines represent the flow of task offloading.

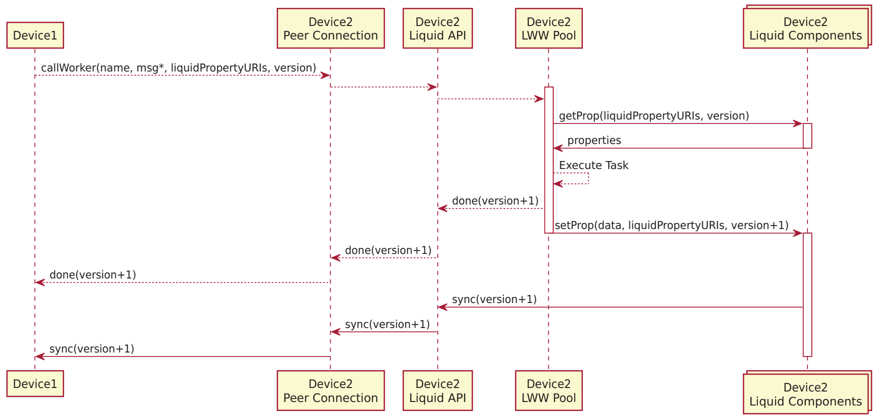


Figure 9 Sequence diagram for asynchronous data transfer.

which transparently will allow the LWW pool to access the data and update it. The task input and result will be automatically synchronized among all paired devices.

In Figure 9 we show how we extended the protocol between the two devices with the asynchronous data transfer. In the synchronous version discussed in Section 3 the payload of the message (*msg*) contains all the data needed by the LWW to execute the task, now that we rely on the data synchronization of the Liquid.js framework, *msg** contains only the data that is not stored and synced by the Liquid.js framework. For the rest of the data that has to be consistently synchronized between the devices, we pass the liquid property URIs referencing the location of the input/output data. Since data is synchronized asynchronously with respect to the task offloading, we cannot guarantee that when the remote device receives the task offloading request, it also already holds the latest version of the corresponding input data. For this reason, whenever the device offloads a task, it also needs to specify which version of the liquid property the remote device needs to use in order to begin the task execution. If there is at least one *liquidPropertyURI*, then the LWW pool will load the state of the liquid property and pass it to the WebWorker. Once the execution finishes, the remote device immediately notifies the other device that it finished executing the task. The message includes the URI and the new version of the updated liquid property. If any liquid property changed during the execution, these will also be automatically synchronized to make the task execution result accessible across all paired devices. Again, the task completion notification and its output propagation happen asynchronously.

What are the advantages of this approach? In the first place messages exchanged between the devices while performing the task offloading are smaller as they carry a reference to the data vs. the actual data, meaning that communication between the two LWW pools is faster. Additionally, developers can access to two distinct events: *executionEnd* and *data-Synchronized*, these two events can help the developers to report the current status of the application to the user, or they can be used to queue new executions as soon as they are finished. Nevertheless if the data resulting from the offloaded computation has to be sent to the original device, this will require to wait until the liquid properties values have been synchronized. Since Liquid.js data synchronization was developed using the Yjs [28] library, only incremental changes are sent, which results in less data to be sent. More in detail, whenever a JavaScript object property is modified, only the changed property is synchronized, while in the synchronous mode, a copy of the whole object needs to be transferred. Additionally, repeated task executions over the same input data can be offloaded without repeatedly transferring the same data along with each offloading request.

4.4 Task Offloading Policies

Policies are needed for making the liquid WebWorker pool able of automatically deciding where to execute tasks. This could be achieved by feeding the *policy* component with predefined rules selected by the developers of the liquid Web application to, e.g., trade-off energy consumption vs. performance.

Policy rules can impact in the overall execution time of an application and the developer needs to be able to enable or disable some rules depending on the context of the application they are building.

- **Battery status** [21] – in the Web Browser it is possible to gain access to the battery status of a device by using the HTML5 Battery Status API.[4] With this API it is possible to detect whether a device is currently charging or how much charge is left in its battery. The policy rule can exploit this API to prioritize plugged-in devices over battery-supported devices. Tasks would be offloaded to devices with an higher charge level, which would decrease the energy consumption of devices with a low battery level.
- **Privacy or security constraints** – the users of a liquid application can interact with devices they do not directly own. Whenever the users interact with shared or public devices, they have to be aware that they are connected to other people's devices. In any situation where the users interact with any device they do not own, the developer should make sure that the users private data is not sent to a stranger device. The policy rule can decide to send data only to the devices they whitelisted, or to any device owned by a whitelisted user. Similarly, the users should be protected from receiving tasks from devices they do not trust.
- **Device types** – As a heuristic, when lacking additional information, the offloading decision can be based on expectations on the performance of a device by knowing its device type, such as *Desktop*, *Laptop*, *Tablet*, *Phone*. Our policy rule would for example assume a desktop computer to be faster than a smartphone. However, this is only a heuristic.

[4]https://developer.mozilla.org/en-US/docs/Web/API/Battery_Status_API

Another way to infer the type of device from within a Web browser relies on checking the size of the screen or the user-agent property, which however may be changed by the users and would not give any direct evidence about the performance of a device [40]. Precise information about the underlying hardware is unfortunately hidden from within the Web browser. Thus, classifying device by their type only may result in incorrect offloading decisions and should be complemented with, e.g., a benchmark or some statistics over some probe task execution times as described in Section 4.1.

- **Communication and Computation Time** – the policy component should consider the exchanged **data size**, the available **bandwidth** (both upload and download because it could be asymmetric) and the **latency** between the devices into the decision. This policy rule makes offloading decisions based on Equation 1, where the communication time is defined in Equation 2.

$$Communication_{time} + Computation_{time}^{remote} \leq Computation_{time}^{local} \tag{1}$$

$$\begin{aligned} Communication_{time} = {} & (Data_{size}^{out}/Bandwidth_{upload}) \\ & + (Data_{size}^{in}/Bandwidth_{download}) \\ & + (2 \cdot Latency_{time}) \end{aligned} \tag{2}$$

While the $Data_{size}^{in}$ and the network parameters (Bandwidth and Latency) can be measured before taking the offloading decision,[5] the size of the result and the computation time may only be estimated or learned based on the characteristics of the LWW script and the history of its past executions.

The advanced features we described in this section allow to simplify some of the terms of the inequality. With the micro-benchmark results we can attempt to predict beforehand which

[5]The $Latency$ can be measured while the devices exchange the connection handshake. The $Bandwidth_{upload}$ and $Bandwidth_{download}$ can be estimated only after some large messages are exchanged between the two peers as unfortunately the Network Information API (https://developer.mozilla.org/en-US/docs/Web/API/Network_Information_API) is not advanced enough to return the exact values for the upload and download speed, but it can only describe the type of connection, e.g. *wifi*, *cellular*.

device has the lower $Computation_{time}^{remote}$ and $Computation_{time}^{local}$. If the remote time is lower, then we can analyze what is the communication cost of the offloading process.

With the asynchronous data synchronization we try to lower the size of $Data_{size}^{in}$ and $Data_{size}^{out}$. In the best case scenario, where all the data is already stored on the remote device, $Data_{size}^{in}$ and $Data_{size}^{out}$ become negligible as only a reference to the data is sent.

5 Scenarios

Liquid WebWorkers can be used to improve the performance of liquid Web applications in simultaneous usage scenarios featuring the opportunity to offload local computations to remote devices owned by the same user or by multiple users (as long as the users trust one another and are willing to share their CPU/energy resources).

Within the simultaneous use case scenario we distinguish two usage categories:

- **Unrestricted** – In this category the environment is composed only by devices that volunteer to freely share computations with each other. All the devices agree that they trust all other devices and they can offload computations freely. The devices will also attempt to execute all offloaded tasks whenever they receive them and promise to return valid results.
- **Restricted** – In this category the connected devices only offer limited access and a lower degree of trust, where they cannot always execute or exchange tasks between each other. Devices can be restricted from executing or offloading tasks for multiple reasons, e.g.:
  - privacy – in order to guarantee data privacy in multi-users scenarios, the application may restrict to offload tasks to devices owned by strangers. In this case, the offloading will take place only among devices of the same owner. When users bring only one single device to run the collaborative application, this device can be prevented to offload tasks to others devices, meaning it has to execute all of them locally;

– security – arbitrary LWWs migration and execution can be used to push malware to the neighbour connected devices and they can be used to execute malicious tasks on other users devices. LWW policies can be implemented in such a way they limit the offload execution of certain tasks (identified by their names) and thus prevent the execution of unknown tasks on unaware devices.
– application dependencies – restrictions can also be programmed to satisfy application specific requirements or dependencies, e.g. in an IoT scenario only some kind of devices is entitled to receive offloaded tasks, because the tasks would need to access some specific sensor attached to the device.

More in general, we distinguish between **push** or **pull** restrictions. A specific device can be restricted from pushing tasks that need to be offloaded on other connected devices, or a specific device can be blocked from pulling tasks which have been offloaded from other devices. In the extreme case, it could be possible that a device can only offload tasks to other devices without ever accepting to run tasks offloaded from other devices (or viceversa).

5.1 Simultaneous Use – Single User Scenario

5.1.1 Editors (e.g. image processing)

LWWs can be used to speed up the process of applying computationally intensive image filters on the pictures displayed in a multi-device Web application (see Figure 10). The liquid application is meant to operate on three different devices:

- **a smartphone**, which is used to take pictures through the integrated camera sensor;
- **a tablet**, which is used to browse the pictures and is used to select which filters should be applied to the images;
- **a laptop or a computer**, which is used to display the pictures on a big screen.

The three devices run the same application simultaneously and they share the pictures between each other, e.g. whenever a picture is taken on the smartphone, it is copied across all other devices.

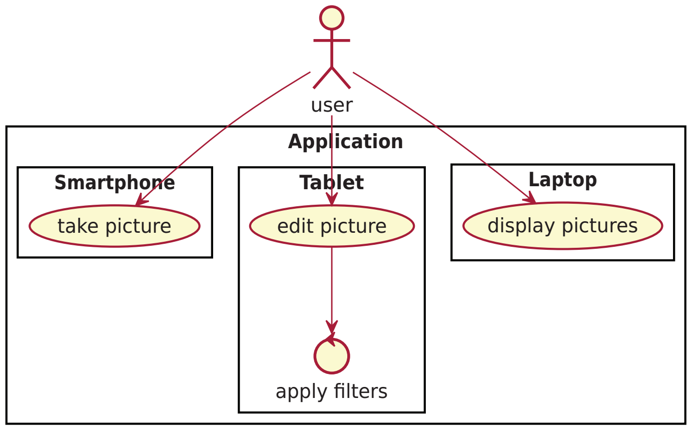


Figure 10 Liquid image processing scenario.

Without LWWs the tablet is in charge of computing and executing all possible edit operations selected by the user, while the smartphone and the laptop would be idle most of the time as they would only serve as input/output devices.

When LWWs are activated, the devices are able to offload computations among one another. In this case the tablet does not have to be burdened with all the image processing tasks, but also the smartphone and the laptop can participate with the goal of improving the overall response time of the image filtering feature of the application. In this particular scenario all the devices are owned by the same user and they are *unrestricted*, that means that any device can freely accept all incoming offloading requests and it can forward them to any other device.

5.1.2 Public displays

In Figure 11 we show a scenario where a single user runs an application on multiple devices, however not all of the devices are owned by the user. In this case the user owns the smartphone, while the public display is owned by the city, which deployed it outside of a train station. The display can be used by anyone by scanning a QR-code printed on the frame of the screen so that the encoded URL is opened on their device mobile Web browser. Once the phone is connected to the screen, the user

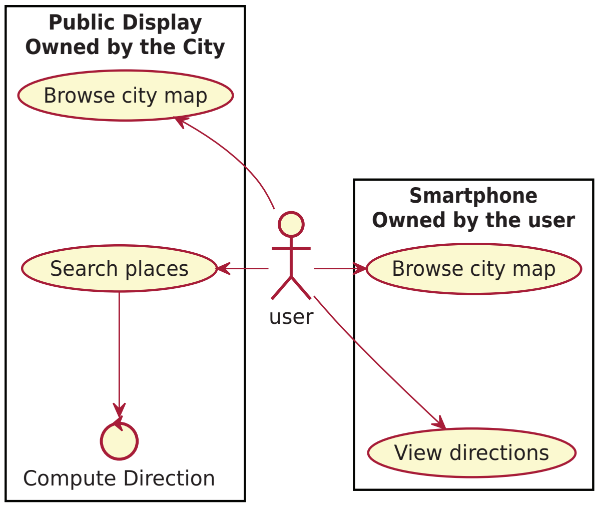


Figure 11 Public display scenario with liquid WebWorkers.

can search on the displayed map for any interesting place in the city, or even compute the shortest path to a given location. While searching for a building is not a complex operation, computing the shortest path may take some time and, since the display should always be responsive to the user interaction, the computation for the shortest path is offloaded to the smartphone. In this use case scenario there is a clear trade-off between the execution time of the algorithm and the bandwidth required to send the map to the smartphones. In the case that the display owners consider the execution time on the display more costly than the bandwidth usage, then they will prefer to offload the execution on the smartphones, even when the phone CPU is weaker than the one on the smart display. They are likely to choose the asynchronous data synchronization, as it will cache the map on the smartphones and it will make it available on the devices for multiple consecutive computations of the shortest route.

While users wait for the task completion, they can browse for other locations or even queue up new computations on their smartphone. Once the requested shortest path tasks are computed, the solutions are stored directly on the phone and the user may display them at any time, even if the smartphone is not connected to the public display anymore.

5.2 Simultaneous Use – Multiple Users Scenarios

5.2.1 Education – teaching programming

In this multi-user scenario we have two user categories: the students and the professors. The professors run the application on their own computers, while the students can access it with their laptops, or even with their tablets or smartphones [39] (see Figure 12). The professors can create new questions at any time, e.g. "*transform this for loop into a while loop*" or similar programming-related questions. The students can see the questions and they can answer by sending a piece of code to the professor. The professors can then choose to display any received answer, they can edit the answers if they spot some errors and then they can display the result of the code execution returned by re-running the code. In order to display the result, the professors need to execute the code, which may lead to three main problems:

- the execution never finishes or it takes to long to finish;
- the code is malicious and tries to block the professor's computer;

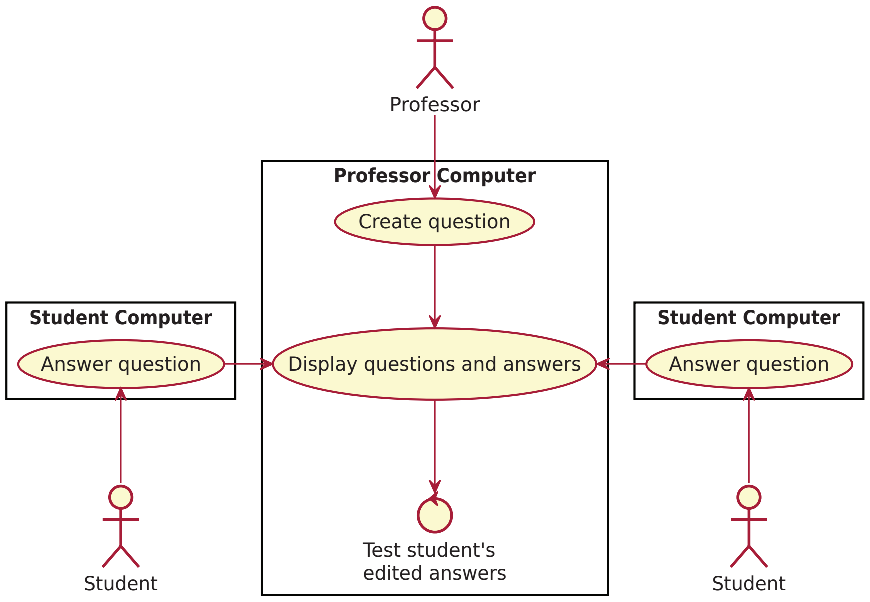


Figure 12 Education scenario with liquid WebWorkers.

- the code is malicious and attempts to corrupt the data contained in the professor's browser storage, e.g. it tries to display on the screen some private data, or it tries to communicate with external Websites;

Without LWWs the code submitted by students must be executed on the professor device, with all the risks to execute buggy or malicious code that can stop the application or disrupt the lesson taught by the professor. In this scenario, LWWs are useful to offload the computation on the computer that originally sent the answer. In this case the professor computer usage is restricted, because it does not accept any incoming execution request, but it always offloads them to the students' computers.

6 Evaluation

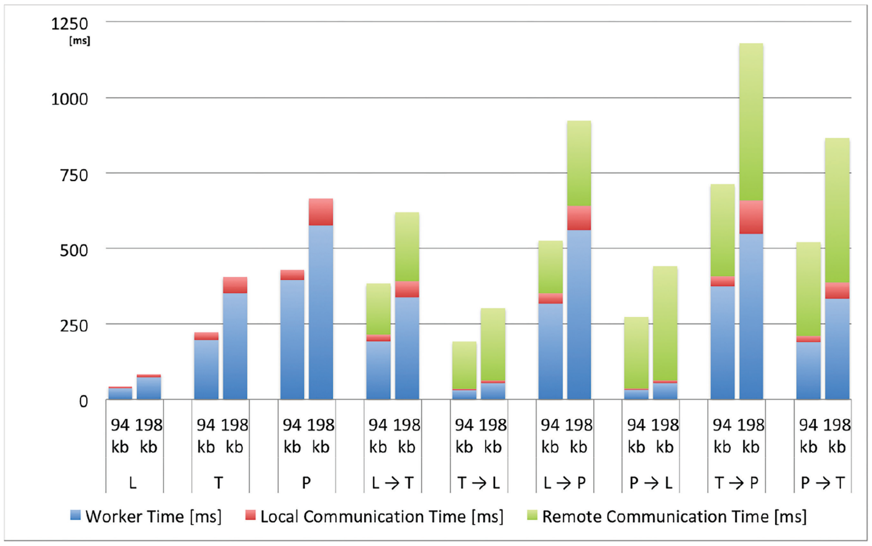
In order to study the feasibility and performance of the liquid Web-Worker concept, in this section we present the results of an evaluation of the Liquid.js prototype implementation.

6.1 Test Scenario: Offloading Image Processing Tasks

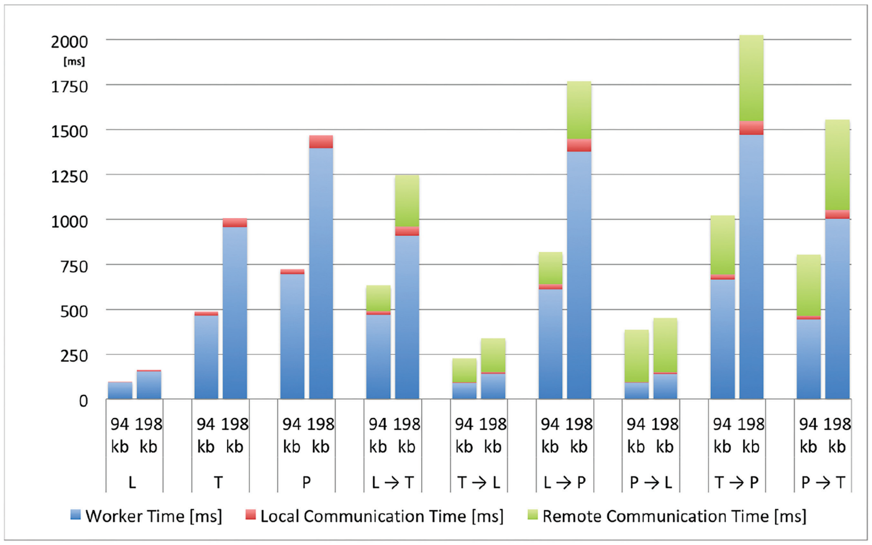
The Liquid.js framework comes with various demo applications, including the liquid camera. This allows users to take pictures with their devices' Webcams, share pictures and display them across multiple devices, and apply a variety of image transformation filters. Applying filters to the images displayed on one device will immediately show the result on all copies of the image found across all connected devices. Since filtering images is a CPU-intensive operation, we have migrated the existing implementation based on WebWorkers to use the LWW pool. Figures 13 and 14 show the results of our preliminary experiments using LWWs.

6.2 Testbed Configuration

All experiments described hereafter are ran using different machines connected to the same private WiFi 5GHz network with the following hardware and OS specification:

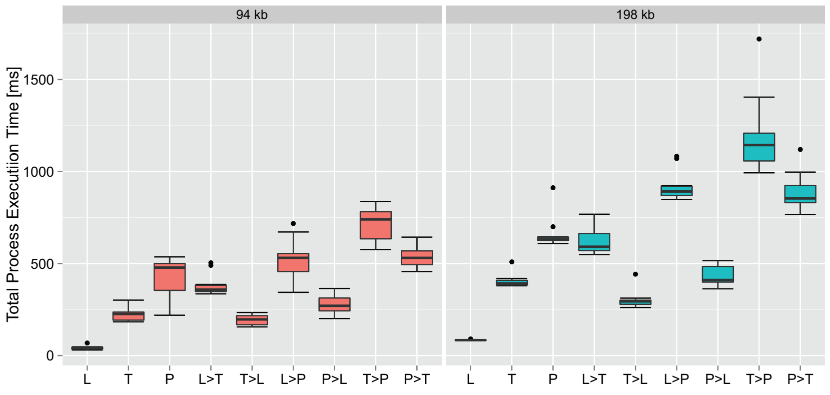


(a) Edge Detection Workload

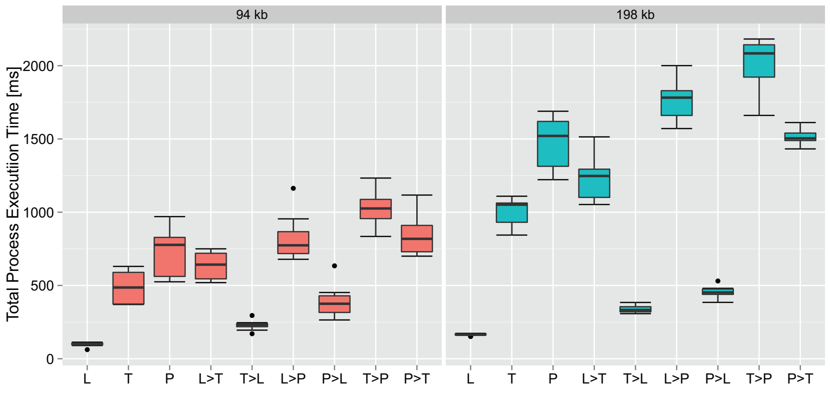


(b) Improved Edge Detection Workload

Figure 13 Average Processing and Communication Time of the liquid WebWorkers offloaded across different pairs of devices (**L**, Laptop, **T**, Tablet, **P**, Phone) [13].



(a) Edge Detection Workload



(b) Improved Edge Detection Workload

Figure 14 Boxplots of the Total Process Execution Time of the LWW offloaded across different pairs of devices (**L**, Laptop, **T**, Tablet, **P**, Phone) [13].

- **Laptop (L):** MacBook Pro (Retina, 15-inch, Mid 2014), 2.2 GHz Intel Core i7, macOS High Sierra Version 10.13.2, Chrome Version 64.0;
- **Tablet (T):** Samsung Galaxy Tab A (2016), Octa Core 1.6 GHz, Android Version 7.0, Chrome Version 64.0;
- **Phone (P):** Samsung J5 (2015), Quad Core 1.2 GHz, Android Version 5.1.1, Chrome Version 62.0.

In this study we show the performance for all shown configurations given the three different kind of devices. The policy loaded inside the LWW takes the decision not to or to offload the execution to other devices based on a predefined static configuration used to explore all possible device combinations in the experiments.

6.3 Workloads

In this evaluation we run two different experiments by applying various filters to the same picture. In the first "Edge Detection" experiment (Figure 13(a)) we apply to the image the Sobel operator filter (using a 3×3 convolution matrix kernel). In the second "Improved Edge Detection" experiment (Figure 13(b)) we improve the result of the edge detection by chaining multiple filters. Compared to the first, the second experiment puts a larger workload on the device CPUs as they run multiple filters with larger kernels. The chained filters are:

1. a sharpening filter implemented by using a convolution filter with a 5×5 kernel;
2. an embossing filter using a 5×5 kernel;
3. the Sobel operator filter using a 5×5 kernel.

For each experiment we apply the filter on two different image resolutions, consequently changing the size of the message exchanged between devices. Both versions of the image are encoded using the *PNG* format and are transferred with messages of size **94196 bytes** and **198560 bytes**.

6.4 Measurements

Each experiment was ran 10 times, during each trial we applied the filters 25 times for both image sizes for all different device offloading combinations. Between two trials we reset the execution environment by restarting the Web browser on all devices. The values of the execution time shown in Figure 13 are computed as the average over the 10 trials.

6.5 Results

The charts show the average time spent by the devices in order to execute a submitted task. Using three different colors we highlight the time elapsed during (see Equation 3): the *worker processing time* in blue, the *remote (or cross-device) communication time* in green, and the *local (or intra-device) communication time* in red. The *worker time* represents the time spent running the LWW script to process the submitted task; the *remote communication time* is spent during the transfer of the submitted task and its output result between the local and remote devices; the *local communication time* includes the time for sending and receiving back the task from the main thread to the LWW, the time employed for message marshalling and unmarshalling, the time spent idle in a message queue, and the overhead of the logging needed to gather performance data for this evaluation.

$$\begin{aligned} Process_{time}^{total} = {} & PromisePreProcess_{time} \\ & + Send_{time}^{offload} + MessageQueue_{time} \\ & + WorkerExecution_{time} + Marshalling_{time} \\ & + Send_{time}^{response} + PromisePostProcess_{time} \end{aligned} \tag{3}$$

6.5.1 Edge detection case (Figures 13(a) and 14(a))

The fastest execution happens on the laptop (**L**) without any offloading. The laptop finishes the process on average about five times faster than the tablet (**T**), and nine times faster than the phone (**P**) for both image sizes. It is interesting to see that every time the laptop was configured to offload work to any other device (**L**→**T**, and **L**→**P**), the overall execution took longer due to the slower *worker processing time* of the remote devices and the additional *remote communication time* required to transfer the task and the response between the devices; the same behavior can be observed when the tablet offloads its work to the phone (**T**→**P**).

In the **T**→**L** and **P**→**L** offloading configurations, the overall execution is faster when compared with the local execution without offloading cases. The elapsed *worker time* of the laptop is so low compared to the one of the tablet and the phone that, despite the penalty due to the remote communication time, the total execution time remains lower. **T**→**L** is on average 81% faster than **T** and **P**→**L** is on average 64% faster than **P**. Despite the expectation that also the configuration **P**→**T** would execute faster than **P**, this was not observed because of the *communication time*. So there were no benefits in offloading the task from the phone to the tablet, in fact in this case the performance worsened.

As a side note, we observed that the WiFi data transmission performance depends on the device, with the phone's available bandwidth being smaller than on the other devices. This behavior is evident when comparing all offloading configurations where the phone is involved with all other configurations. In particular the communication time between the phone and the tablet is double than the time between the laptop and the tablet. This could also be caused by the physical proximity of the devices during the tests which may have led to some interference as indicated by changes of the WiFi signal strength on the devices. We did not attempt to shield the devices to reduce measurement noise because our goal was to reproduce real-world usage conditions.

From this experiment we can conclude that it is possible to benefit from using LWWs and thus it is possible to lower the total processing time by offloading tasks to nearby devices. However, this can be achieved only when the extra communication overhead is smaller than the gained processing time due to the faster remote CPU.

6.5.2 Improved Edge Detection case (Figures 13(a) and 14(b))

In this experiment we stress the devices more as we increase the workload exerted on the LWWs. On average the *worker processing time* for this experiment is 248% longer on all devices when compared to the previous experiment.

We can observe that the *local communication time* is unaffected by the experiment, but the average *remote communication time* slightly changes due to the previously discussed noisy WiFi channel.

Offloading computations to the phone never registers lower process execution times (**L**→**P** and **T**→**P**), which is the conclusion we observed before.

Particularly interesting in the second experiment are the values registered in configuration **P**→**T** compared to values registered in **P**. In this case we observe that again on average **P** is slightly faster (82–85ms difference) than **P**→**T**. Still, if we examine the trend by including the data from the experiment before we can see that the longer the *worker time*, the better it is to offload workload from **P** to **T**. Eventually, for heavy workloads, offloading to a tablet would be better than executing the tasks on the phone, because the *remote communication time* remains mostly constant for the same image size while the *worker time* constitutes the dominant factor.

6.6 Micro-Benchmark Evaluation

Can the micro benchmarking score accurately predict the capabilities of a connected device? We answer this question by comparing the scores returned by the test benchmark against the results obtained in the previous sections.

Figure 15 shows the results obtained by running the benchmark a total of 200 times for each device. The benchmark is executed at startup and then it is repeated periodically every *300 seconds*. The application is restarted after it has completed 50 benchmarks, meaning that the application runs continuously for 15000 seconds (approximately 4 hours), before the device is restarted. In order to reduce the measurements noise, during the benchmark execution the user does not interact with the device, simulating a comparable scenario with the previous evaluation.

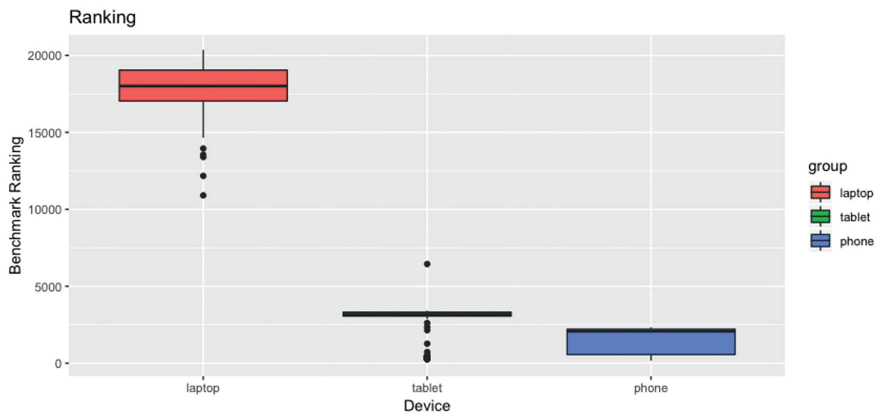


Figure 15 Boxplot of the benchmark scores for each device.

Table 3 Average Benchmark Ranking and Average Processing

	94kb				198kb					
	Comb		**Sobel**		**Comb**		**Sobel**		**Benchmark**	
	[ms]	%	[ms]	%	[ms]	%	[ms]	%	Score	$\%^{-1}$
laptop	91	19.5	33	16.8	145	15.2	60	17.3	17898	15.1
tablet	466		196		953		345		2707	
laptop	91	13.7	33	9.1	145	10.3	60	10.5	17898	8.7
phone	662		363		1413		569		1550	
tablet	466	70.4	196	54.0	953	67.4	345	60.6	2707	57.3
phone	662		363		1413		569		1550	

As expected the score for the laptop is higher than the other devices, with an average score of *17898*, while the tablet scores *2707.3* and the phone *1550.4*.

In Table 3 we compare the average worker execution times against the benchmark scores. We list all possible pairs of devices and compute both the ratio between their respective average worker execution times and their average score returned by the benchmark. In *yellow*, *orange*, and *red* we highlight the average ratios computed for the same couple of devices. Since the machines do not change, we expect that the execution ratios do not change within the same pair even if the experiment is different. Whenever the LWW executes a longer task on a machine, then we expect it proportionally increases also on the other one. In all three couples, we see that the average ratio between the sampled bencharked ratio and the real world example are similar. The benchmarked ratio for *taptop-tablet* differs on average the *13%* of the real world scenario ratio, the *laptop-phone* ratio differs on average the *20%* from the real world ratio, and the *tablet-phone* ratio only *9%*.

Figure 16 shows how much time it takes to execute the benchmark on each device. The execution time is stable, with very few outliers on the tablet device. On average, between all three the devices, it takes *9.3 seconds* to execute the micro-benchmark.

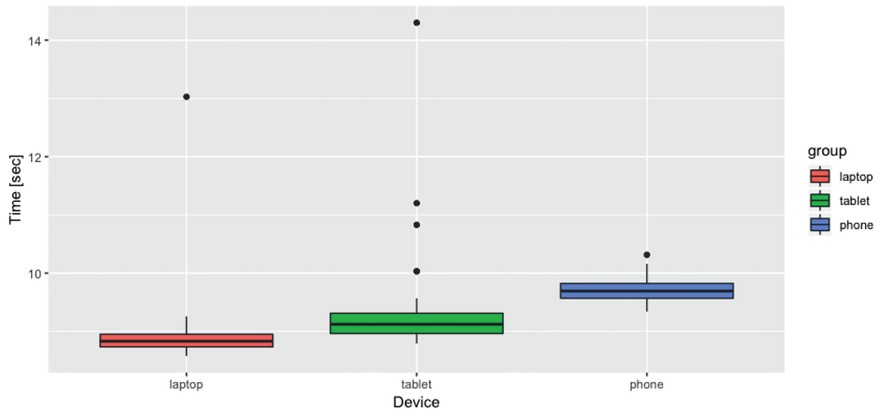


Figure 16 Boxplot of the benchmark execution times.

7 Conclusions

The HTML5 standards and the latest evolution of Web technologies are shifting the center of Web application towards the edge. As Web applications are growing more and more complex, and as users interact with more and more devices simultaneously, we need to understand how devices can support each other in a distributed mobile Web environment. As described in this paper, horizontal task offloading with liquid WebWorkers can benefit both single and multi-user real-world use case scenarios.

More in detail, in this paper we presented the design of liquid WebWorkers and their implementation within the Liquid.js framework. LWWs are designed for building liquid application featuring heavy computations which are dynamically redeployed across multiple, partially-idle heterogeneous devices. The goal is to avoid slowing down the overall performance of the application because some of its components are running on a slow, bottleneck device. The preliminary evaluation of the liquid WebWorkers concept and our prototype implementation shows that there is the opportunity for increasing the overall performance of a liquid Web applications when LWW are migrated from slow to more powerful devices.

Overall, with the extension of Liquid.js presented in this paper, we can create complex device ensembles able to directly connect and transparently share storage and computation resources between them. This allows developers to control how their liquid Web applications trade off performance against energy consumption.

8 Future Work

8.1 Failure Handling – Automatic Timeout Configuration

In Section 4.2 we discussed how to handle errors thrown in the remote execution process. When we discussed the timeout case we stated that the timeout should be configured by the developer of the application. The timeout value directly depends on the expected execution time of the remote peer, which completely depends on the task it executes. There is no default timeout value that satisfies all possible applications, meaning that the developer should know beforehand what is the expected timeout value for his application. The worker execution time may depend on the power of the device, the size of the data and the complexity of the algorithm ran inside the task, and can be difficult to predict it for the developer. In the future we plan to understand how to predict the value for the timeout, in such a way that the developers do not have to configure it if they do not want to set a constant value.

8.2 Stateful LWWs

The current LWW programming model simplifies the HTML5 Web-Worker model to run stateless computations, where each task can be independently re-assigned to a different device. We plan to extend LWWs to support stateful workers exchanging an arbitrary number of messages during arbitrary computations, making the transparent migration of such workers more challenging. This can be solved by reusing existing liquid storage facilities of Liquid.js that have been originally designed to migrate and synchronize stateful Web components.

Acknowledgements

This work is partially supported by the SNF with the “Fundamentals of Parallel Programming for PaaS Clouds” project (Nr. 153560).

References

[1] D. P. Anderson and G. Fedak. The Computational and Storage Potential of Volunteer Computing. In *Cluster Computing and the Grid. CCGRID 06*, volume 1, pages 73–80. IEEE, 2006.

[2] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. In *Computer Networks*, volume 54, pages 2787–2805, Elsevier, 2010.

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog Computing and Its Role in the Internet of Things. In *Proc. of the MCC workshop on Mobile cloud computing. MCC'12*, pages 13–16. ACM, 2012.

[4] M. Bynens and J. Dalton. *Benchmark.js v2.1.0*. https://bench markjs. com/, 2016

[5] S. Clinch. Smartphones and Pervasive Public Displays. *IEEE Pervasive Computing*, volume 12(1), pages 92–95, IEEE, 2013.

[6] D. J. Cook and S. K. Das. How Smart are Our Environments? An Updated Look at the State of the Art. *Pervasive and mobile computing*, volume 3(2), pages 53–73, Elsevier, 2007.

[7] M. S. Corson, R. Laroia, J. Li, V. Park, T. Richardson, and G. Tsirtsis. Toward Proximity-Aware Internetworking. *IEEE Wireless Communications*, volume 17(6), pages 26–33, IEEE, 2010.

[8] R. Cushing, G. H. H. Putra, S. Koulouzis, A. Belloum, M. Bubak, and C. De Laat. Distributed Computing on an Ensemble of Browsers. *IEEE Internet Computing*, volume 17(5), pages 54–61, IEEE, 2013.

[9] H. T. Dinh, C. Lee, D. Niyato, and P. Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing*, volume 13(18), pages 1587–1611, Wiley Online Library, 2011.

[10] G. H. Forman and J. Zahorjan. The Challenges of Mobile Computing. *Computer*, volume 27(4), pages 38–47, IEEE, 1994.

[11] A. Gallidabino and C. Pautasso. Deploying Stateful web Components on Multiple Devices with Liquid.js for Polymer. In *Proc. of Component-Based Software Engineering. CBSE'16*, pages 85–90. IEEE, 2016.

[12] A. Gallidabino and C. Pautasso. Maturity Model for Liquid Web Architectures. In *Proc. of International Conference on Web*

Engineering. ICWE'17, volume 10360, pages 206–224, Springer, 2017.

[13] A. Gallidabino and C. Pautasso. Decentralized Computation Offloading on the Edge with Liquid WebWorkers. In *Proc. of International Conference On Web Engineering. ICWE'18)*, volume 10845, pages 145–161, Springer, 2018.

[14] A. Gallidabino and C. Pautasso. The Liquid User Experience API. In *Proc. of the International Conference on the World Wide Web. WWW'18*, pages 767–774, International World Wide Web Conferences, 2018.

[15] A. Gallidabino, C. Pautasso, T. Mikkonen, K. Systa, J.-P. Voutilainen, and A. Taivalsaari. Architecting Liquid Software. In *Journal of Web Engineering*, volume 16(5&6), pages 433–470, Rinton Press, 2017.

[16] Google. The new multi-screen world: Understanding cross-platform consumer behavior. http://services.google.com/fh/files/misc/multiscreenworld_final.pdf, 2012.

[17] Google. The connected consumer. http://www.google.com.sg/publicdata/explore?ds=dg8d1eetcqsb1_, 2017.

[18] D. Guinard, V. Trifa, F. Mattern, and E. Wilde. From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices. In *Architecting the Internet of Things*, pages 97–129. Springer, 2011.

[19] J. Hartman, U. Manber, L. Peterson, and T. Proebsting. Liquid Software: A New Paradigm for Networked Systems. Technical Report 96–11, University of Arizona, 1996.

[20] M. Hirsch, J. M. Rodríguez, C. Mateos, and A. Zunino. A Two-Phase Energy-Aware Scheduling Approach for CPU-Intensive Jobs in Mobile Grids. In *Journal of Grid Computing*, volume 15(1), pages 55–80, Springer, 2017.

[21] M. Hirsch, J. M. Rodriguez, A. Zunino, and C. Mateos. Battery-Aware Centralized Schedulers for CPU-Bound Jobs in Mobile Grids. In *Pervasive and Mobile Computing*, volume 29, pages 73–94, Elsevier, 2016.

[22] K. Huppler. The Art of Building a Good Benchmark. In *Proc. of Technology Conference on Performance Evaluation and Benchmarking. TPCTC'09*, pages 18–30. Springer, 2009.

[23] W.-J. Ke and S.-D. Wang. Reliability Evaluation for Distributed Computing Networks with Imperfect Nodes. In *IEEE Transactions on Reliability*, volume 46(3), pages 342–349, IEEE, 1997.

[24] C. P. Kruger and G. P. Hancke. Benchmarking Internet of Things Devices. In *Proc. of International Conference on Industrial Informatics. INDIN'14*, pages 611–616, IEEE, 2014.

[25] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava. A Survey of Computation Offloading for Mobile Systems. In *Mobile Networks and Applications*, volume 18(1), pages 129–140, Springer, 2013.

[26] S. W. Loke, K. Napier, A. Alali, N. Fernando, and W. Rahayu. Mobile Computations with Surrounding Devices: Proximity Sensing and MultiLayered Work Stealing. In *ACM Transactions on Embedded Computing Systems (TECS)*, volume 14(2), article 22, ACM, 2015.

[27] T. H. Luan, L. Gao, Z. Li, Y. Xiang, G. Wei, and L. Sun. Fog Computing: Focusing on Mobile Users at the Edge. In *Networking and Internet Architecture*, arXiv preprint arXiv:1502.01815, 2015.

[28] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma. Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types. In *Proc. of International Conference on Web Engineering. ICWE'15*, pages 675–678. Springer, 2015.

[29] S. Okamoto and M. Kohana. Load Distribution by Using Web Workers for a Real-Time Web Application. In *International Journal of Web Information Systems*, volume 7(4), pages 381–395, Emerald Publishing, 2011.

[30] E. Pitoura and B. Bhargava. Maintaining Consistency of Data in Mobile Distributed Environments. In *Proc. of Distributed Computing Systems*, pages 404–413. IEEE, 1995.

[31] S. Poslad. *Ubiquitous Computing: Smart Devices, Environments and Interactions*. John Wiley & Sons, 2011.

[32] M. Satyanarayanan. The Emergence of Edge Computing. In *Computer*, volume 50(1), pages 30–39, 2017.

[33] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. In *IEEE Personal communications*, volume 8(4), pages 10–17, IEEE, 2001.

[34] M. Satyanarayanan, J. J. Kistler, L. B. Mummert, M. R. Ebling, P. Kumar, and Q. Lu. Experience with Disconnected Operation in a Mobile Computing Environment. In *Mobile Computing*, pages 537–570. Springer, 1993.

[35] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Mobile Computing Systems and Applications, 1994. Proceedings., Workshop on*, pages 85–90. IEEE, 1994.

[36] W. Shi and S. Dustdar. The Promise of Edge Computing. In *Computer*, volume 49(5), pages 78–81, IEEE, 2016.

[37] A. Taivalsaari and T. Mikkonen. A Roadmap to the Programmable World: Software Challenges in the IoT Era. In *IEEE Software*, volume 34(1), pages 72–80, IEEE, 2017.

[38] A. Taivalsaari, T. Mikkonen, and K. Systa. Liquid Software Manifesto: The Era of Multiple Device Ownership and Its Implications for Software Architecture. In *Proc of Computer Software and Applications Conference. COMPSAC'14*, pages 338–343, IEEE, 2014.

[39] V. Triglianos. *ASQ: Active Learning with Interactive Web Presentations and Classroom Analytics*. Phd Thesis, USI, Lugano, Switzerland, 2018.

[40] R. Upathilake, Y. Li and A. Matrawy. A Classification of Web Browser Fingerprinting Techniques. In *Proc. of International Conference on New Technologies, Mobility and Security. NTMS'15*, pages 1–5, IEEE, 2015.

[41] E. Welbourne, L. Battle, G. Cole, K. Gould, K. Rector, S. Raymer, M. Balazinska, and G. Borriello. Building the Internet of Things Using RFID: the RFID Ecosystem Experience. In *IEEE Internet computing*, volume 13(3), pages 48–55, IEEE, 2009.

[42] A. Whitmore, A. Agarwal, and L. Da Xu. The Internet of Things – A Survey of Topics and Trends. In *Information Systems Frontiers*, volume 17(2), pages 261–274, Springer, 2015.

[43] S. Xinogalos, K. E. Psannis, and A. Sifaleras. Recent Advances Delivered by HTML 5 in Mobile Cloud Computing Applications:

A Survey. In *Proc. of Balkan Conference in Informatics*, pages 199–204. ACM, 2012.

[44] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of Things for Smart Cities. In *IEEE Internet of Things journal*, volume 1(1), pages 22–32, IEEE 2014.

Biographies

Andrea Gallidabino. He is part of the Architecture, Design and Web Information Systems Engineering Group under the supervision of his advisor Prof. Cesare Pautasso. The group is part of the Software Institute at Univeristà della Svizzera Italiana. As a researcher in the group he focuses his work mainly on liquid software and real-time communication Web technologies. Moreover he helps the professor with lectures, interacting with students on a daily basis. His research interests are: Web technologies, real-time applications, liquid software, and multi-device interactions. You can find more information on http://www.inf.usi.ch/phd/gallidabino/ and follow him @AGallidabino.

Cesare Pautasso. He is a Full Professor at the Software Institute of the Faculty of Informatics, USI Lugano, Switzerland, and formerly researcher at the IBM Zurich Research Lab (2007) and senior researcher at ETH Zurich (2004–2007) where he completed his Ph.D. in 2004. At USI he leads the Architecture, Design and Web Information Systems Engineering research group. He is currently supervising the research of a group of Ph.D. students building experimental systems to explore the intersection of cloud computing, software architecture, Web engineering, and business process management, with ongoing projects exploring workflow benchmarking, RESTful conversations, and liquid software. He is the coauthor of the book *SOA with REST* (2012) and currently writing a book titled *Just Send an Email: Anti-Patterns for Email-Centric Organizations* (published on LeanPub). He is coeditor of the IEEE Software Insight department and general chair of the 16th International Conference on Web Engineering (ICWE2016). You can find more information on http://www.pautasso.info and follow him @pautasso@scholar.social.