Decentralized Computation Offloading on the Edge with Liquid WebWorkers

Andrea Gallidabino and Cesare Pautasso

Software Institute, Faculty of Informatics, USI Lugano, Switzerland
{name.surname}@usi.ch

Abstract. Liquid Web applications seamlessly flow across any kind of device ranging from powerful desktop and laptop devices to smaller devices, such as tablets, smart phones or any device capable of running a Web browser. In this context, there is the opportunity to reduce the execution time of CPU-intensive tasks or limit their energy consumption by offloading them across the set of machines running the liquid Web application. To do so, in this paper we present Liquid WebWorkers, which build upon the standard HTML5 WebWorker API and transparently offload the task execution to other devices and manage the corresponding data transfer. This way, Web developers can reuse their existing WebWorker scripts without any changes. We present how to create a pool of paired devices and compare different policies for choosing the target device that have been implemented in the Liquid.js framework.

Keywords: WebWorkers, Edge Computing, Liquid Software

1 Introduction

Nowadays 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 [9]. As the average number of devices per user increases, this affects the way users interact with their software applications running across the set of devices they own [8]. The Liquid Software paradigm [11,20] empowers users to run their applications on a set of multiple heterogeneous devices. In this paper we focus on the parallel screening scenario where one or multiple users run applications across multiple devices at the same time. While we have previously discussed how the liquid Web application may be designed to take advantage of all available devices [7], in this paper we focus on the computational aspects of the application. Users interacting with multiple devices may trigger data synchronization activities that will ensure a consistent view over the state of the distributed Web application is presented. 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 present Liquid WebWorkers, a novel abstraction built on top of the standard HTML5 WebWorker API[1], which allow developers to add

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

parallelism to their liquid Web applications by offloading computational tasks from a device to another with the goal of executing CPU-intensive tasks across the most suitable user-owned device. This will contribute to speed up the overall computation of the response that later will be propagated across all paired devices.

2 Related Work

The technological foundation of Liquid Software emerges from the Internet of Things [1], the Web of Things [10], or more in general the Programmable World [19]. Users live inside an adaptive ecosystem composed of all smart objects surrounding them [21]; whenever a smart object enters or leaves the proximity of the user, the liquid software automatically grows or shrinks, adapting itself to the new set of devices.

Edge computing [18] research 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 [17]. The optimization reduces bandwidth consumption and latency in the communication between the edge devices, making it possible to reduce overall processing time of an operation. Fog computing [2,15] 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 [16] literature.

Traditionally the distributed computational resources was given mainly by server centric clusters of computers or the cloud, however it is demonstrated that the Web, by employing Web browsers and WebWorkers, is ready to be a decentralized computation platform [3] as well. While most existing computational offloading work focuses on shifting workloads from mobile devices to the Cloud [4], in this paper we study how to use 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.* [12] 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 devices during an initial centralized decision-making phase. This is followed by the job stealing phase, when 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 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 a devices hardware and software configuration is not always accessible.

Loke *et al.* [14], propose a similar system allowing multi-layered job stealing techniques also with a hybrid approach (both centralized and decentralized) to offload decision making.

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 be potentially be 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 input tasks should be executed and reliably dispatch tasks towards the corresponding LWW, 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 return 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 pinged LWW is not yet instantiated or there was an internal error in the LWW execution.

Liquid WebWorker Pool API Table 1 lists all methods exposed by the API and its constructor. The Liquid WebWorker pool can be instantiated by passing the reference to a *sendMessage* function whose signature must accept

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

Table 1: Liquid WebWorker Pool API

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)

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

The pool exposes eight methods:

– **createWorker:** instantiate a new LWW and bind it to the LWW pool automatically. The pool may contain any number of workers, limited only by the memory available to the Web browser. 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 *URL* pointing to a Web resource, or it can be a String containing the actual script. 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 the information about all devices. Depending on the policy rules employed this object may contain different information (see subsection 6.1).

– **removePairedDevice:** this method removes a paired device from the stored list of paired devices.

– **callWorker:** the callWorker is used to submit a task into the pool, which later will be executed either locally or remotely. Once submitted the pool decides where the task will be ran, then it creates the corresponding promises and calls the sendMessage function if the task is executed remotely, otherwise it will call the _callWorker function.

Table 2: Liquid WebWorker API

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 is used to submit a task into the pool which is forced to run locally. This method directly pushes the task message into the right LWW instance and waits for its asynchronous response by setting up a promise object.

– **forwardMessage:** whenever a device receives a message sent from the sendMessage function, it must be forwarded inside the pool so that it can be executed on the remote device. The developer must forward those messages into the LWW 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.

Liquid WebWorker API Table 2 lists all methods exposed by the LWW and its constructor. If an invalid or undefined LWW pool is passed as parameter of the constructor, then the methods *callWorker* and *_callWorker* will behave equivalently and the Liquid WebWorker will not offload the execution on remote devices. That is because, without a connected pool, the LWW cannot poll it and ask where the submitted tasks should be executed. The LWW is stateless, thus it does not store information about paired devices nor it knows if it is paired to other LWWs.

The 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:

– **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.

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

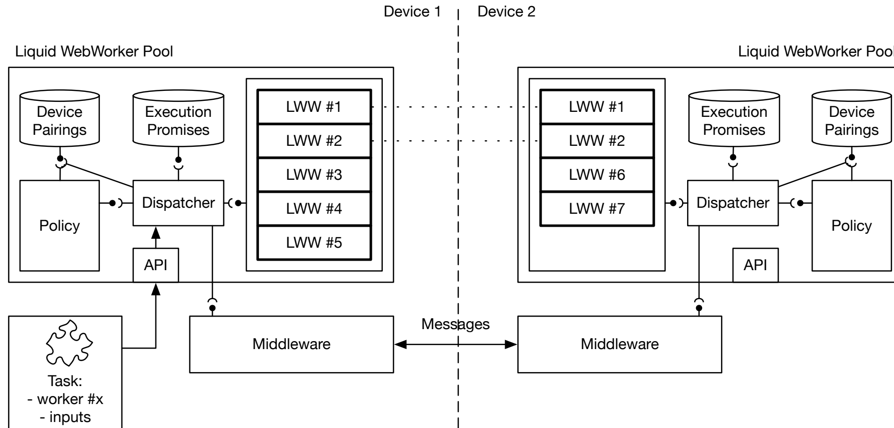


Fig. 1: Liquid WebWorkers Architecture. Arrows show the flow of a task and the exchange of messages between clients. Dotted lines indicate *paired* relationships between Liquid WebWorker instances.

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 return the results by posting the response and its unique identifier. The promise contains the callback that must be fulfilled or rejected when the remote device or the local worker respond. 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.

– **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 save 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). 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.

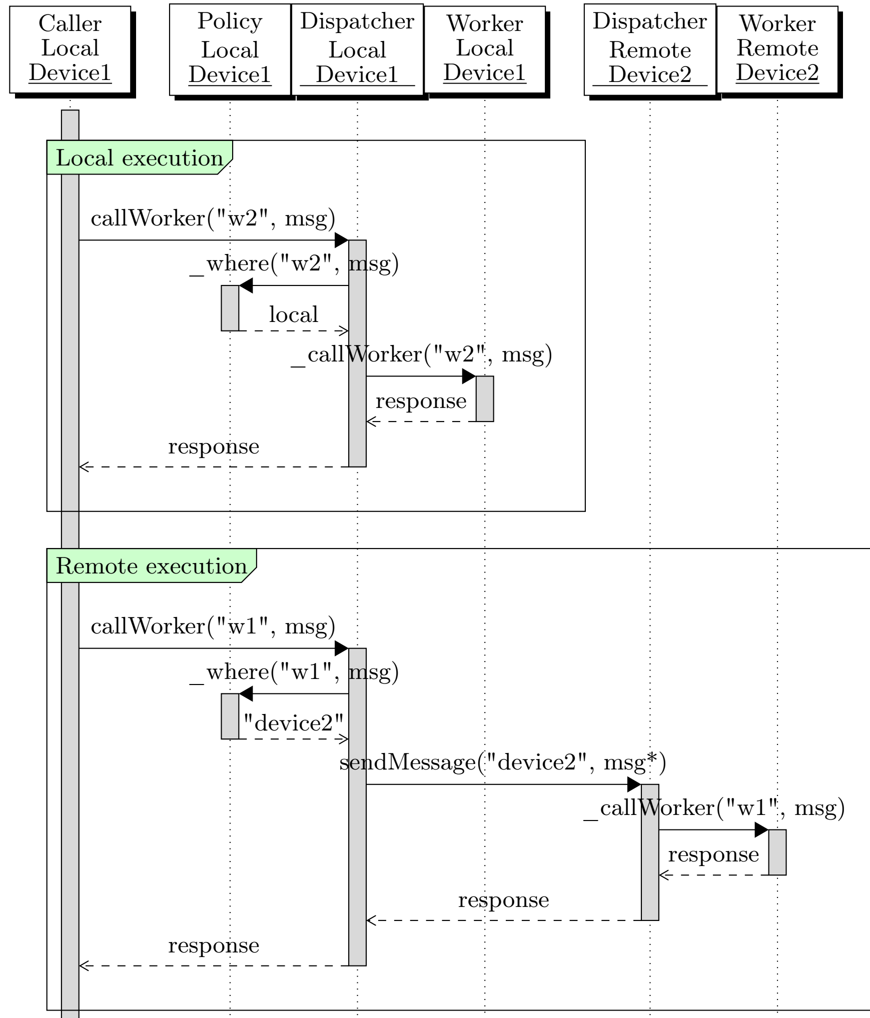


Fig. 2: Liquid WebWorker local and remote execution sequence diagram

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

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

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 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.

3.3 Liquid.js Prototype

We built a Liquid Web Workers prototype within the Liquid.js for Polymer [5] 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 illustrates a simplified component view of both Liquid.js extended with the LWW pool. The Liquid WebWorker pool is managed by the framework itself hidden behind its own API [6]. 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.

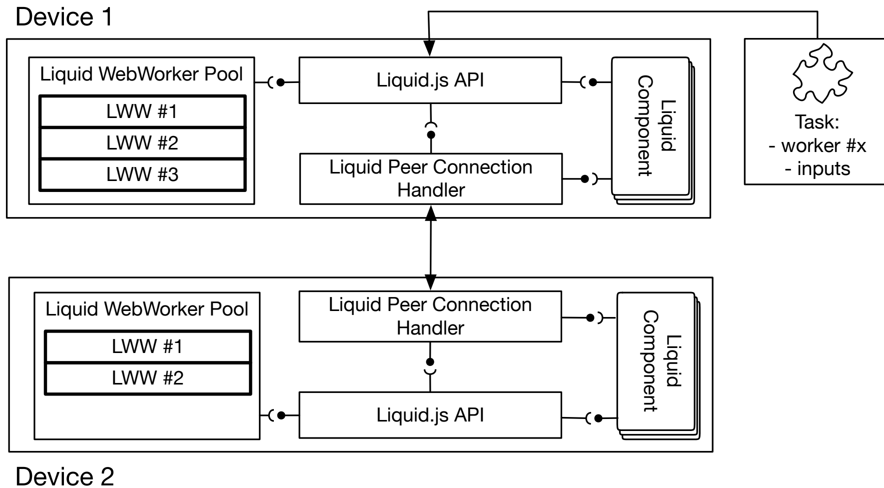


Fig. 3: Component view of the implementation of Liquid WebWorkers inside the Liquid.js for Polymer framework.

4 Evaluation

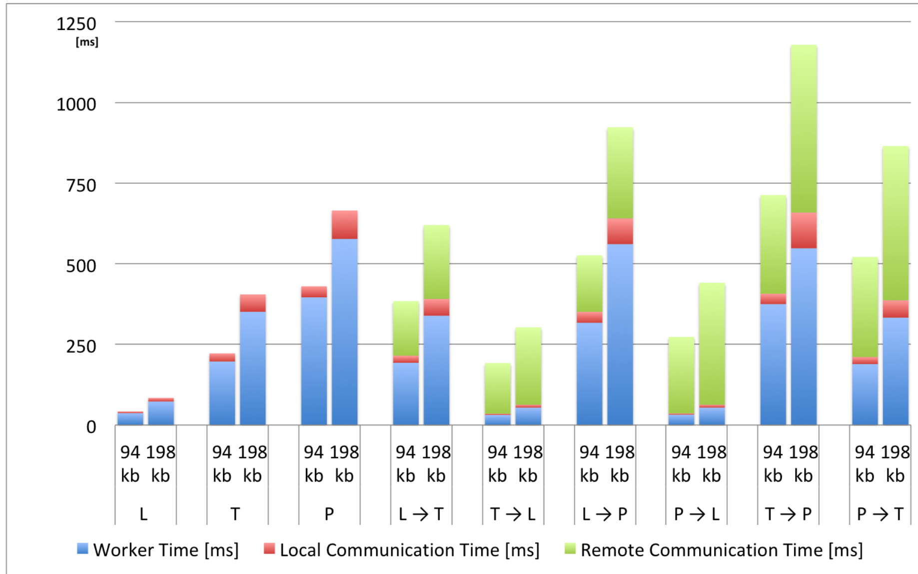
In order to study the feasibility and performance of the Liquid WebWorker concept, in this Section we present the results of an evaluation of the Liquid.js prototype implementation.

4.1 Test Scenario: Offloading Image Processing Tasks

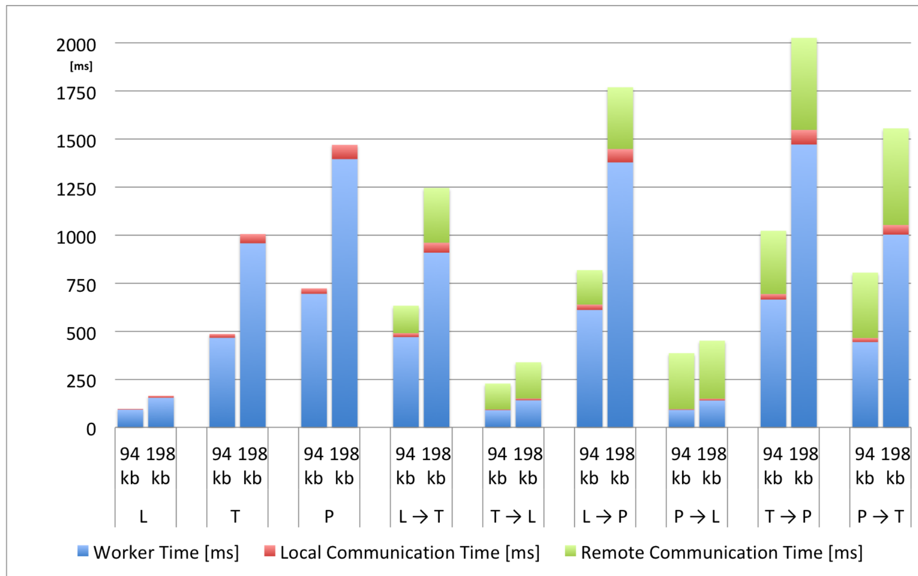
The Liquid.js framework comes with various demo applications, including the liquid camera. This allows users to take pictures with their devices' Webcams, share pictures and display them across multiple devices, and apply a variety of image transformation filters. Applying filters to the images displayed 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 Web workers to use the LWW pool. Figure 4 and 5 show the results of our preliminary experiments using LWWs.

4.2 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: – **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.



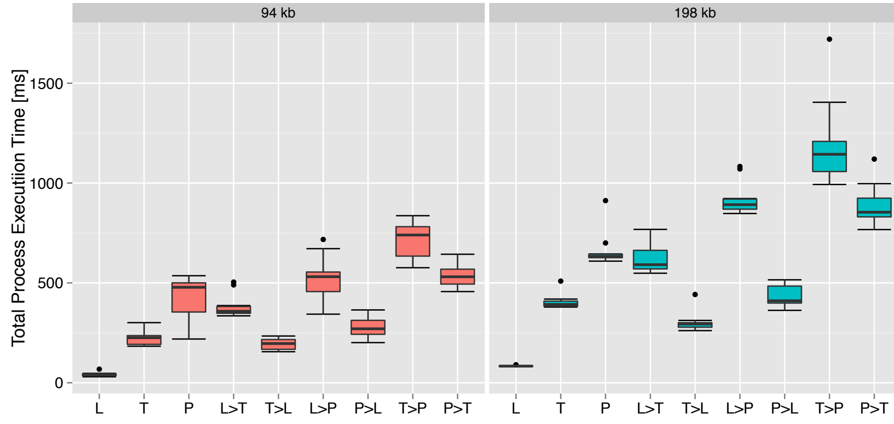
(a) Edge Detection Workload



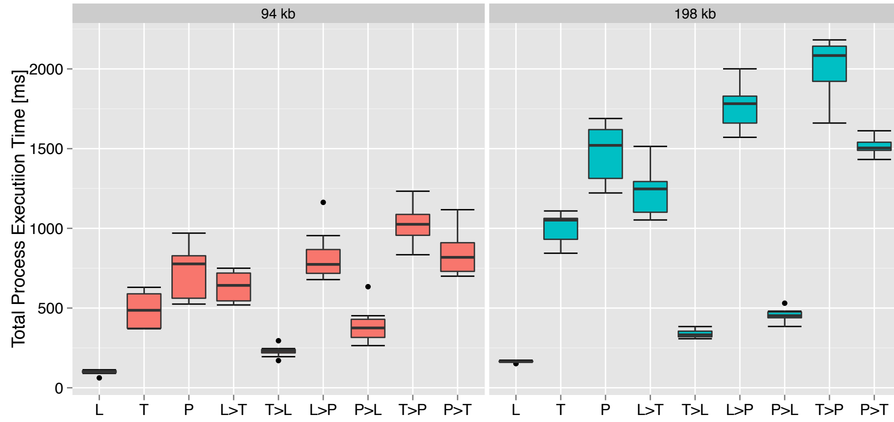
(b) Improved Edge Detection Workload

Fig. 4: Average Processing and Communication Time of the Liquid WebWorkers offloaded across different pairs of devices (**L** Laptop, **T**, Tablet, **P** Phone)

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 prede-



(a) Edge Detection Workload



(b) Improved Edge Detection Workload

Fig. 5: Boxplots of the Total Process Execution Time of the LWW offloaded across different pairs of devices (**L** Laptop, **T**, Tablet, **P** Phone)

fined static configuration used to explore all possible device combinations in the experiments.

4.3 Workloads

In this evaluation we run two different experiments by applying various filters to the same picture. In the first "Edge Detection" experiment (Figure 4b) we apply to the image the Sobel operator filter (using a 3x3 convolution matrix kernel).In the second "Improved Edge Detection" experiment (Figure 4b) 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 5x5 kernel;2. an embossing filter using a 5x5 kernel; 3. the Sobel operator filter using a 5x5 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**.

4.4 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 4 are computed as the average over the 10 trials.

4.5 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 1): 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.

$$Process_t^{total} = PromisePreProcess_t + Send_t^{offload} + MessageQueue_t + WorkerExecution_t + Marshalling_t + Send_t^{response} + PromisePostProcess_t \quad (1)$$

Edge Detection case (Figure 4a and 5a): 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.

Improved Edge Detection case (Figure 4b and 5b): 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.

5 Conclusions

In this paper we presented the design of Liquid WebWorkers and their implementation within the Liquid.js framework. LWWs are suited for building liquid application featuring heavy computations dynamically redeployed across multiple, partially-idle heterogeneous devices. The goal is to avoid slowing down the overall performance of the application because of some 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.

6 Future Work

6.1 Policy Rules

We plan to make the Liquid WebWorkers 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.

In this paper we demonstrated how such rules could impact the overall execution time. These policy rules need to be further investigated in order to determine how to provide them with enough information about the state of the various devices. For example: **battery levels** [13] would use the current state of charge to prioritize plugged-in devices over battery-supported devices, or **privacy constraints** could prevent the usage of public or shared devices and limit which devices would receive privacy-sensitive user data only shared among user-owned devices.

During the experiments, the decision of where to offload the computation was precomputed to test all possible device type combinations. The obtained results can help to make the decision automatic and adaptive, capable of taking the right decision with the goal to reduce the overall execution time. Here we propose two policy rules derived by the behavior observed during the experiments.

Device type as initial priority - Some device are better than others and we can initially have some expectation on the performance of a device by knowing its device type in advance, such as *Desktop*, *Laptop*, *Tablet*, *Phone*. Our policy rule would have some expectation on the device type, for example we **expect** that a desktop computer would be faster than a smartphone, however this is only an heuristic. In general, each device (and Web browser) performance would need to be benchmarked. Unfortunately, detailed hardware specifications are not directly available from a Web browser, making it difficult to determine accurately its performance a priori.

Lower Communication 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 rule tries to optimize Equation 2 where the communication time is defined as written in Equation 3.

$$Communication_t + Computation_t^{remote} \leq Computation_t^{local} \quad (2)$$

$$\begin{aligned} Communication_t = (Data_{size}^{out} * Bandwidth_{upload}) + \\ (Data_{size}^{in} * Bandwidth_{download}) + (Latency_t * 2) \end{aligned} \quad (3)$$

While the $Data_{size}^{in}$ and the network parameters (Bandwidth and Latency) can be measured before taking the offloading decision, 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.

6.2 Stateful LWWs

The current LWW programming model simplifies the HTML5 WebWorker 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. Atzori, L., Iera, A., Morabito, G.: The internet of things: A survey. Computer networks 54(15), 2787–2805 (2010)
2. Bonomi, F., Milito, R., Zhu, J., Addepalli, S.: Fog computing and its role in the internet of things. In: Proc. of the first edition of the MCC workshop on Mobile cloud computing. pp. 13–16. ACM (2012)
3. Cushing, R., Putra, G.H.H., Koulouzis, S., Belloum, A., Bubak, M., De Laat, C.: Distributed computing on an ensemble of browsers. IEEE Internet Computing 17(5), 54–61 (2013)
4. Dinh, H.T., Lee, C., Niyato, D., Wang, P.: A survey of mobile cloud computing: architecture, applications, and approaches. Wireless communications and mobile computing 13(18), 1587–1611 (2013)
5. Gallidabino, A., Pautasso, C.: Deploying stateful web components on multiple devices with liquid.js for Polymer. In: Proc. of CBSE. pp. 85–90. IEEE (2016)

6. Gallidabino, A., Pautasso, C.: The liquid user experience API. In: Proc. of the 27th International Conference on the World Wide Web (WWW) (2018)
7. Gallidabino, A., Pautasso, C., Mikkonen, T., Systa, K., Voutilainen, J.P., Taivalsaari, A.: Architecting liquid software. Journal of Web Engineering 16(5&6), 433–470 (September 2017)
8. Google: The new multi-screen world: Understanding cross-platform consumer behavior. http://services.google.com/fh/files/misc/multiscreenworld_final.pdf (2012)
9. Google: The connected consumer. http://www.google.com.sg/publicdata/explore?ds=dg8d1eetcqsb1_ (2015)
10. Guinard, D., Trifa, V., Mattern, F., Wilde, E.: From the internet of things to the web of things: Resource-oriented architecture and best practices. In: Uckelmann, D., Harrison, M., Michahelles, F. (eds.) Architecting the Internet of Things, pp. 97–129. Springer (2011)
11. Hartman, J., Manber, U., Peterson, L., Proebsting, T.: Liquid software: A new paradigm for networked systems. Tech. Rep. 96-11, University of Arizona (1996)
12. Hirsch, M., Rodríguez, J.M., Mateos, C., Zunino, A.: A two-phase energy-aware scheduling approach for cpu-intensive jobs in mobile grids. Journal of Grid Computing 15(1), 55–80 (2017)
13. Hirsch, M., Rodriguez, J.M., Zunino, A., Mateos, C.: Battery-aware centralized schedulers for cpu-bound jobs in mobile grids. Pervasive and Mobile Computing 29, 73–94 (2016)
14. Loke, S.W., Napier, K., Alali, A., Fernando, N., Rahayu, W.: Mobile computations with surrounding devices: Proximity sensing and multilayered work stealing. ACM Transactions on Embedded Computing Systems (TECS) 14(2), 22 (2015)
15. Luan, T.H., Gao, L., Li, Z., Xiang, Y., Wei, G., Sun, L.: Fog computing: Focusing on mobile users at the edge. arXiv preprint arXiv:1502.01815 (2015)
16. Poslad, S.: Ubiquitous computing: smart devices, environments and interactions. John Wiley & Sons (2011)
17. Satyanarayanan, M.: The emergence of edge computing. Computer 50(1), 30–39 (2017), https://doi.org/10.1109/MC.2017.9
18. Shi, W., Dustdar, S.: The promise of edge computing. Computer 49(5), 78–81 (2016)
19. Taivalsaari, A., Mikkonen, T.: A roadmap to the programmable world: Software challenges in the IoT era. IEEE Software 34(1), 72–80 (Jan/Feb 2017)
20. Taivalsaari, A., Mikkonen, T., Systa, K.: Liquid software manifesto: The era of multiple device ownership and its implications for software architecture. In: 38th Computer Software and Applications Conference (COMPSAC). pp. 338–343 (2014)
21. Welbourne, E., Battle, L., Cole, G., Gould, K., Rector, K., Raymer, S., Balazinska, M., Borriello, G.: Building the internet of things using rfid: the rfid ecosystem experience. IEEE Internet computing 13(3) (2009)