Tuning Browser-to-Browser Offloading for Heterogeneous Stream Processing Web Applications

Masiar Babazadeh

Faculty of Informatics, University of Lugano (USI), Switzerland
{name.surname@usi.ch}

Abstract. Software that runs on the Cloud may be offloaded to clients to ease the computational effort on the server side, while clients may as well offload computations back to the cloud or to other clients if it becomes too taxing on their machines. In this paper we present how we autonomically deal with browser-to-browser operator offloading in Web Liquid Streams, a stream processing framework that lets developers implement streaming topologies on any Web-enabled device. We show how we first implemented the offloading of streaming operators, and how we subsequently improved our approach. Our experimental results show how the new approach takes advantage of additional resources to reduce the end-to-end message delay and the queue sizes under heavy load.

Keywords: Stream Processing, Peer to Peer Offloading, Autonomic Controller

1 Introduction and Related Work

In recent years the Web has become an important platform to develop any kind of application. With the requirement that applications show results updated in real time, it has become important for developers to use suitable stream processing abstractions and frameworks [1]. In this paper we explore the opportunity to offload part of the computation across different devices on which the stream processing topology is distributed. For example, offloading and migrating part of the computation eases Web servers deployed in the Cloud from computational effort [2], while reducing the response time on clients that can locally perform part of the computation instead of simply rendering the result. Conversely, energy consumption may become a concern and thus the reverse offloading may happen, from clients back to the Cloud [3], or – as we are going to discuss in this paper – to other clients nearby.

In this paper, we take the offloading concept and apply it in a distributed streaming infrastructure [4] where clients and the cloud are tightly coupled to form stream processing topologies built using the Web Liquid Streams (WLS [5]) framework. WLS lets Web developers setup topologies of distributed streaming operators written in JavaScript and run them across a variety of heterogeneous Web-enabled devices.

O. Kopp, J. Lenhard, C. Pautasso (Eds.): 9th ZEUS Workshop, ZEUS 2017, Lugano, Switzerland, 13-14 February 2017, published at http://ceur-ws.org/

WLS can be used to develop streaming topologies for data analytics, complex event processing [6], real-time sensor monitoring, and so on. Makers [7] that want to automate their homes and offices with Web-enabled sensors and microcontrollers can use WLS to deploy full JavaScript applicatons across their home server and house sensors without dealing with different programming languages.

Streaming operators are implemented in JavaScript using the WLS primitives, and are deployed by the WLS runtime across the pool of available devices. Users may let the runtime decide where to distribute the operators, or manually constrain where each operator has to run. This is done through a topology description file which is used to deploy and start a topology. The WLS runtime is in charge to bind the operators using the appropriate channels and start the data stream. Depending on the hosting platform, we developed different communication infrastructure. For server-to-server communication we make use of ZeroMQ, for server-to-browser and browser-to-server we use WebSockets, while for browser-to-browser communication we use the recently developed WebRTC.

WLS implements an autonomic controller in charge to increase or decrease parallelism at operator level by adding WebWorkers in bottleneck situations, and removing them when they become idle. At topology level, the controller parallelizes the execution of operators across multiple devices by splitting the operator, or fusing it back together when bottlenecks are solved, depending on the variable data load [8].

In this paper we focus on the controller running on each Web browser and how it can be tuned to make operator offloading decisions based on different policies and threshold settings. This work is based on our previous work on the distributed controller infrastructure [9].

2 Operator Offloading Controller

The Web Liquid Streams controller deals with bottlenecks and failures by migrating streaming operators on available machines, effectively offloading the work from the overloaded machines.

The controller constantly monitors the topology and its operators in order to detect bottlenecks and/or failures and solve them. In particular, it queries the streaming operators as the topology runs, and by keeping into account the length of the queues, the input and output rates, as well as the CPU consumption, it takes decisions to improve the throughput.

We currently have two distinct implementations of the controller infrastructure: one dedicated to Web server and microcontroller operators (Node.JS implementation) and one running on Web browsers. We decided to have two different implementations because of the different kind of environmental performance metrics we can access. In Node.JS our controller has access to many more details regarding the underlying OS, the available memory, CPU utilization, and network bandwidth. For example, to trigger a migration or an operator split in a Web server, the controller needs to check the CPU utilization of the machine and decides to ask for help when it reaches a given specified threshold [9].

The Web browser controller has only access to a subset of the environment metrics, which also heavily depends on the Web browser being used. Thus, the decision policy needs to be adapted accordingly. In a Web browser environment we only know how many CPUs the machine has available through the `window.navigator.hardwareConcurrency` API. We thus decided to use this number as a cap for the number of maximum concurrent WebWorkers on the Web browser.

2.1 First Iteration of the Controller

The first implementation of the Web browser controller was designed to behave very similarly to its Web server counterpart. All the functionalities related to fault tolerance and load balancing were implemented in the same way using a different approach. While the controller cycle was left at 500 milliseconds per cycle, we applied different approaches given the differences in environment.

To compute the CPU usage we relied on the `hardwareConcurrency` API.

$$P(t) > T_{CPU} * \texttt{hardwareConcurrency}$$

When the number of WebWorker threads P on the machine reached the amount of concurrent CPUs T_{CPU} available on the machine (100% CPU capacity), the controller raised a CPU overload flag to the central WLS runtime, which in turn contacted a free host to make it start running a copy of the overloaded operator, thus parallelizing its execution across different machines.

WLS also support flow control to avoid overfilling queues of overloaded operators. This "slow mode" is also triggered by the controller using a two-threshold rule:

$$\begin{aligned} Q(t) &> Tqh \rightarrow SlowModeON \\ &< Tql \rightarrow SlowModeOFF \end{aligned}$$

The idea behind the slow mode is to slow down the input rate of a given (overloaded) operator to help it dispatch the messages in its queue Q, while increasing the input rate on other instances of said operator. Once the queue is consumed below a given threshold Tql, the controller removes the slow mode, re-enabling the normal stream flow. In [9] we tuned many aspects of the controller, including the slow mode, for three different families of experiments. Results suggested that $Tqh = 20$ messages in the queue were enough to trigger the slow mode, which was released the moment the queue reached $Tql = 10$ or less elements.

2.2 Improving the Controller

By stressing the controller through further experiments we noticed that the metrics given by the `hardwareConcurrency` API were not sufficiently precise to make the controller behave correctly. Very high throughputs and big message sizes in fact showed how the controller took too much time in noticing the

bottleneck and asking for help, resulting in very high delays and big queues. We addressed the problem by tuning some controller parameters. We reduced the cycle of the controller from 500 to 300 milliseconds per cycle. We then halved the threshold to trigger the CPU flag to $T_{CPU} = 50\%$.

Finally, we also halved the thresholds to trigger and release the slow mode while maintaining the same formula, obtaining $Tqh = 10$ and $Tql = 5$.

3 Evaluation

To evaluate the performance improvement of the various Web browser controller configurations, we developed a Web cam streaming application. The proposed topology is a simple three-staged pipeline where the producer (first stage) gathers data from the user's Web cam. The filter (second stage) receives the image, runs a face detection algorithm and draws a decoration over the heads of the detected faces, then forwards the result downstream. This stage is intentionally made heavy in terms of CPU time in order to create bottlenecks and stress the controller. The consumer (third stage) shows the image on screen. All the operators run on a Web browser.

The machines used are three MacBook Pros, one with 16GB RAM 2.3 GHz Intel Core i7 (peer 1), one with 4GB RAM 2GHz Intel Core i7 (peer 2), and one with 4GB RAM 2.53GHz Intel Core i5 (peer 3). All the machines run macOS Sierra 10.12 and running Chrome version 45.0.2454.101 (64-bit). We are using an old Chrome version because of recent restrictions related to the use of WebRTC. The WLS server side runs on a 48-core Intel Xeon 2.00GHz processors and 128GB RAM.

For this experiment we deployed the producer and consumer on peer 3, while used peer 1 (P1) and peer 2 (P2) as free machines where the WLS runtime can run and eventually offload the computation of the filter. By default, the WLS runtime picks the strongest machine (peer 1) to run the filter at the beginning, and subsequently offloads the computation on peer 2. We decided to send a total of 6000 messages for this experiment at the rate of 75 milliseconds per message (about 13 messages per second). The size of the Web cam image is fixed to 400x300 pixels, which are converted into messages to be sent, for a total weight of about 800Kb per message. We used the WiFi (averaging around 200Mbit per second) to simulate a normal use case scenario environment.

For this kind of experiment, we decided to focus on the delay as the main metric to measure. In streaming applications that may take into account real-time sensor data, we want to be able to process this data as soon as possible with as little delay as possible. We make use of the queue size as well to show how, by transitioning from one implementation of the controller to another, we are able to keep the queue size small by parallelizing faster and more efficiently.

The results show three different implementations of the controller (C1, C2, C3): the first one and the third one represent the two controller implementations described in Section 2. The second one shows a compromise between the two, a similar controller with a cycle frequency of 300 milliseconds and with

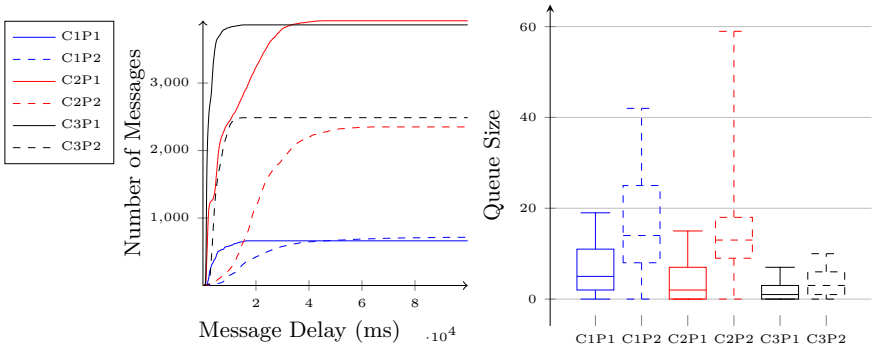

Fig. 1: Message delay and queue size distributions per peer over the three controller configurations.

the concurrency check halved with respect to the number of processes on the hosting machine, but with the first slow mode implementation. Table 1 shows the differences in the three configurations.

Figure 1 shows the distribution of the end-to-end message delay of the three configurations as well as the boxplot (mix-max range, mean and quartiles) of the queue size. We cut the delay axis at 100 seconds delay for readability.

The first configuration shows two curves that are slowly growing in terms of number of messages, while keep increasing in delay. This is given by the fact that with our original configuration, under such circumstances, the controller was too slow to raise a CPU flag to the WLS runtime, while the slow mode was triggered too late. The queue boxplots show they were filled and kept being so, inducing message loss and resulting in this small and slowly increasing curve.

The second configuration shows two distinct curves that correctly processed the messages in the topology and processed the vast majority of messages with a delay of 3 to 35 seconds. Both curves follow a similar trend, one being the peer 1 (strongest machine), processing more messages, while the other being on peer 2. By increasing the frequency of the controller cycle and raising a CPU flag sooner, we are able to deal with the increasing queue sizes, keeping them short, and parallelize the work sooner.

A similar trend can be found in the third configuration, where we notice that the majority of messages is executed with less than 10 seconds delay. Both curves grow with the same shape, keeping the delay lower than the second configuration.

Table 1: Controller configuration parameters

Tuned Parameter	Configuration 1	Configuration 2	Configuration 3
Controller Cycle	500ms	300ms	300ms
T_{CPU}	100%	50%	50%
Tqh	20	20	10
Tql	10	10	5

The further improvement in this configuration shows how, by triggering the slow mode earlier, we are able to keep queues even emptier, and thus lowering the delays of the majority of the messages.

4 Conclusions and Future Work

In this paper we have introduced how we approach browser-to-browser operator offloading in the Web Liquid Streams framework. We described our initial control infrastructure and how we improved it to be more responsive in case of increasing workloads. The experimental evaluation shows the benefits of the approach by demonstrating the improvements on the measured end-to-end message delay and operator queue sizes. By increasing even more the workload we may eventually end up filling the queues and all the processors available on every machine. To solve this problem we are implementing support for load shedding [10] that should be triggered by the controller.

References

1. Hochreiner, C., Schulte, S., Dustdar, S., Lecue, F.: Elastic stream processing for distributed environments. IEEE Internet Computing **19**(6) (2015) 54–59
2. Wang, X., Liu, X., Huang, G., Liu, Y.: Appmobicloud: Improving mobile web applications by mobile-cloud convergence. In: Proceedings of the 5th Asia-Pacific Symposium on Internetware. Internetware '13, ACM (2013) 14:1–14:10
3. Banerjee, A., Chen, X., Erman, J., Gopalakrishnan, V., Lee, S., Van Der Merwe, J.: Moca: A lightweight mobile cloud offloading architecture. In: Proceedings of the Eighth ACM International Workshop on Mobility in the Evolving Internet Architecture. MobiArch '13, ACM (2013) 11–16
4. Golab, L., Özsu, M.T.: Data Stream Management. Synthesis Lectures on Data Management. Morgan & Claypool Publishers (2010)
5. Babazadeh, M., Gallidabino, A., Pautasso, C.: Decentralized stream processing over web-enabled devices. In: 4th European Conference on Service-Oriented and Cloud Computing. Volume 9306., Taormina, Italy, Springer (September 2015) 3–18
6. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. **44**(3) (June 2012) 15:1–15:62
7. Anderson, C.: Makers : the new industrial revolution. Random House Business Books, London (2012)
8. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. ACM Comput. Surv. **46**(4) (March 2014) 46:1–46:34
9. Babazadeh, M., Gallidabino, A., Pautasso, C.: Liquid stream processing across web browsers and web servers. In: 15th International Conference on Web Engineering (ICWE 2015), Rotterdam, NL, Springer (June 2015)
10. Gedik, B., Wu, K.L., Yu, P.S., Liu, L.: Adaptive load shedding for windowed stream joins. In: Proceedings of the 14th ACM International Conference on Information and Knowledge Management. CIKM '05, New York, NY, USA, ACM (2005)