Liquid Stream Processing Across Web Browsers and Web Servers

Masiar Babazadeh[✉], Andrea Gallidabino, and Cesare Pautasso

Faculty of Informatics, University of Lugano (USI), Lugano, Switzerland
{masiar.babazadeh,andrea.gallidabino,cesare.pautasso}@usi.ch

Abstract. The recently proposed API definition WebRTC introduced peer-to-peer real time communication between Web browsers, allowing streaming systems to be deployed on browsers in addition to traditional server-side execution environments. While streaming applications can be adapted to run on Web browsers, it remains difficult to deal with temporary disconnections, energy consumption on mobile devices and a potentially very large number of heterogeneous peers that join and leave the execution environment affecting the quality of the stream. In this paper we present the decentralized control approach followed by the Web Liquid Streams (WLS) framework, a novel framework for streaming applications running on Web browsers, Web servers and smart devices. Given the heterogeneity of the deployment environment and the volatility of Web browsers, we implemented a control infrastructure which is able to take operator migration decisions keeping into account the deployment constraints and the unpredictable workload.

1 Introduction

Real-time Web applications which display live updates resulting from complex stream processing topologies have recently started to impact the Web architecture. Web browsers are no longer limited to synchronous request-response HTTP interactions but can now establish full-duplex connections both with the server (WebSockets) and even directly with other browsers (WebRTC). This opens up the opportunity to shift to the Web browser more than a simple view to be rendered over the results of the stream, but also the processing of the data stream operators. Similar to ongoing Rich Internet Application trends [1], the idea is to offload part of the computation to clients that are becoming more and more powerful. This however requires the flexibility to adapt operators to run on a highly heterogeneous execution environment, determine which is the most suitable location for their execution, and deal with frequent outages, limited battery capacity and device failures.

More in detail, while building data streaming applications across Web browsers with WebRTC and WebSockets can be a relatively simple task, managing the deployment of arbitrary topologies, disconnections, and errors without a central entity may become increasingly difficult. The volatile nature of Web browsers (whose tabs may be closed at any time) implies that the system should

be able to restart leaving operators on other available machines. At the same time, CPU and memory consumption as well as data flow rate should be managed and regulated when operators become too taxing on the hosting browser. Whenever a host used up all its resources and causes bottlenecks, the hosted streaming operators should be relocated to other available machines. Special care should be taken in presence of portable devices such as smartphones or tablets, whose battery should be saved whenever possible.

In this paper we show how we deal with these problems in the Web Liquid Streams (WLS) framework. WLS allows developers to implement distributed stream processing topologies composed of JavaScript operators deployed across Web browsers and Web servers. Thanks to the widespread adoption of JavaScript, the lingua franca of the Web, it is possible to deploy and run streaming applications on any Web-enabled device: from small microprocessors, passing through smartphones, all the way to large virtualised Cloud computing clusters running the Node.JS [2] framework. The focus of this paper concerns the WLS distributed control infrastructure which is able to deal with faults and disconnections of the computing operators as well as bottlenecks in the streaming topology. Moreover it is in charge of optimizing the physical deployment of the topology by migrating and cloning operators across Web browsers and Web servers to reduce energy consumption and increase parallelism.

The RESTful API of the first version of WLS, which supported distributed stream processing only over Web server clusters has been previously described in [3]. In this paper we build upon the previous results to present the second version of WLS, which makes the following novel contributions:

– A decentralized infrastructure to deploy both on Web servers and Web browsers a graph of data stream operators in order to reduce the end to end latency of the stream while enforcing given deployment constraints.
– A control infrastructure able to deal with bottlenecks in the stream topology.
– A technique for cloning and relocating operators to dynamically parallelize computationally intensive operators, which can make use of volunteer resources and react to unpredictable workload changes.

The rest of this paper is organized as follows: Section 2 discusses related work, Section 3 introduces the Web Liquid Streams framework while Section 4 describes its control infrastructure. Section 5 shows a preliminary evaluation of the control infrastructure and Section 6 concludes the paper.

2   Related Work

Many stream processing frameworks have been proposed over the years [4,5]. Here we discuss the ones which inspired our work on WLS. Storm [6] is a distributed real-time computational environment originally developed by Twitter. Stream operators are defined by custom created "spouts" and "bolts", used to allow distributed processing of streaming data. Storm presents a two-fold controller infrastructure to deal with worker failure and node failure. Storm only features a server-side deployment, while we extend this concept in WLS by

including Web browsers as part of the target deployment infrastructure and use standard Web protocols to send and receive data.

D-Streams [7] is a framework that provides a set of stream-transformations which treat the stream as a series of deterministic batch computations on very small time intervals. In this way, it reuses the fault tolerance mechanism for batch processing, leveraging MapReduce style recovery. D-Stream features a reconfiguration-style recovery for faults, which was inspirational for our controller; nonetheless, the D-Streams topology cannot be changed and adapted to the workload at runtime. Another related framework is MillWheel [8] by Google. MillWheel helps user build low-latency data-processing applications at large-scale without the need to think about how to deploy it in a distributed execution environment. Our runtime works at the same level of abstraction, but targets a more diverse, volunteer computing-style set of provided resources.

With the rapid growth of mobile smart phones, many efforts targeted improving their energy efficiency and explored how to deal with energy-intensive applications by means of Cloud offloading [9]. With approaches like MAUI [10] or ThinkAir [11], programmers annotate part of the code to be offloaded, and the runtime is able to tell if the offloading of the annotated parts could save energy on the client that would remotely invoke the corresponding methods.

Some research effort has been dedicated towards self-adaptation of streaming applications. A self-adapting optimizer has been presented in [12] where the authors introduced an online and offline scheduler for Storm. Another optimization for Storm has been proposed in [13], where the authors present an optimization algorithm that finds the optimal values for the batch sizes and the degree of parallelism for each Storm node in the dataflow. The algorithm automatically finds the best configuration for a topology, avoiding manual tuning. Our work is very similar, yet the premises of having fully transparent Peers are not met, as WLS is unable to access the complete hardware specifications of a machine from the existing Web browser HTML5 APIs. A more general approach in describing streaming optimizations has been taken in [4], where authors suggest a list of stream processing optimizations, some of which have been applied in our framework (i.e., Operator Fission, Fusion or Load balancing).

The concept of liquid software has been introduced in the Liquid Software Manifesto [14], where the authors use it to represent a seamless multi-device user experience where software can effortlessly flow from one device to another [15]. Likewise, in [16] we described an architectural style for liquid Web services, which can elastically scale taking advantage of heterogeneous computing resources. In the presented cases, the liquid quality is applied to the deployment of software components. In this paper we focus our efforts on the stream software connector and how to characterize its liquid behavior.

3   The Web Liquid Streams Framework

The Web Liquid Streams (WLS) framework helps developers create stream processing topologies and run them across Web servers, Web browsers, and smart

devices. This Section introduces the main abstractions provided by WLS and how they can be used to build Topologies of streaming Operators written in JavaScript. To build streaming applications, developers make use of the following building blocks: **Peers** are physical machines that host the computation. Any Web-enabled device, including Web servers, Web browsers or microprocessors is eligible to become a Peer. In a streaming application, **Operators** receive the data, process it and forward the results downstream. Each Operator is associated to a JavaScript file, which describes the processing logic. The **Topology** defines a graph of Operators, effectively describing the data flow from producer Operators to consumers. The structure of the Topology can dynamically change while the stream is running. Peers can host more than one Operator at a time, and Operators on the same Peer can be part of different streaming Topologies. The way Operators and Peers are organized is dynamic as the number of Operators and their resource usage can change at runtime. Operators can be redundantly deployed on multiple Peers for increased scalability and reliability.

When implementing an Operator, developers do not need to worry about communication protocols as it is the WLS runtime's task to deploy the Operators on the physical hosts based on the Topology description and deployment constraints, and thus taking care of abstracting away the actual stream communication channels. To do so, the runtime includes a control infrastructure able to deal with disconnections and load fluctuations by means of Operator migration.

3.1   Liquid Stream Processing

We use the Liquid Software metaphor [14–16] to visualize the properties of the flow of data elements through the Operators that are deployed over a large and variable number of networked Peers owned by the users of the platform. The nature of the Peers can be heterogeneous: big Web servers with a lot of storage and RAM, Web browsers running on smart phones or personal computers, but also Web-enabled microcontrollers and smart devices such as Raspberry PI or Tessel.IO. Users of the system can share their own resources in a cooperative way by either connecting to the system through a Web browser or a Web server. WLS is able to handle the churn of connecting and leaving Peers through a gossip algorithm [17] , which propagates across all Peers the information about connected Peers where Operators can be deployed.

As a liquid adapts its shape to the one of its container, the WLS framework adapts the operators and the stream data rate to fit within the available resources. When the resource demand increases (i.e., because of an increase in the stream rate, or the processing power required to process some stream elements), new resources are allocated by the controller. Likewise, these will be elastically de-allocated once the stream resource demand decreases.

3.2   Operator Cloning and Migration

We implemented two basic mechanisms on the Operators: cloning and migration [18]. Operator cloning is the process of creating an exact copy of the

Operator on another Peer to solve bottlenecks by increasing the parallelism. The process can be reversed once the bottlenecks are solved and there is no need to keep more than one Peer busy.

Migration is the process of moving one Operator from one Peer to another at runtime. Migration in WLS can be triggered when a Peer expresses its willingness to leave the system, or when the battery level of a device reaches a predefined threshold. In that case, the Operators are migrated on other Peers to prevent disconnection errors when the battery is completely drained out. During the migration, the controller creates copies of these Operators on the designated machines and performs the bindings to and from other Operators, given the Topology structure. Considering [18], WLS performs a re-initialization of the relocated code on the new execution environment.

4   Decentralized Control Infrastructure

By uploading a file containing the Topology description through the WLS API [3], users of the system can deploy and execute their Topologies on a subset of the available Peers, which will be autonomously and transparently managed by WLS. Topologies can also be created manually through the command-line interface of the system or through a graphical monitoring tool. Upon receiving a Topology description or manual configuration commands, the system checks if the request can be satisfied with the currently available resources. If that is the case, WLS allocates and deploys Operators on designated Peers and starts the data stream between them. In this section we present the autonomic control infrastructure that automatically deploys Topologies and sends reconfiguration commands through the same WLS API. Each Web server Peer features its own local controller which deals with the connections and disconnections of the Web browser Peers attached to it as well as the parallelisation of Operators running on it (by forking Node.js processes). A similar controller is deployed on each Web browser Peer, where it parallelises the Operator execution spawning additional Web Workers. Likewises it monitors the local environment conditions to detect bottlenecks, battery shortages or disconnections and to signal them to the corresponding controller on the Web server. This establishes a hierarchical control infrastructure whereby each Operator is managed by a separate entity, coordinated by the controller responsible for managing the lifecycle and regulating the performance of the whole Topology.

4.1   Deployment Constraints

The control infrastructure deals with disconnections of Peers, load fluctuations and Operator migrations. In this paper we target the ability of the controller to deploy, migrate, and clone Operators across Peers seamlessly in order to face disconnections, but also to improve the overall performance of the Topology in terms of latency and energy consumption. To do so, the controller takes into account a list of constraints that are specified in the Topology description. The

constraints are presented here in descending order of importance: i) **Hardware Dependencies**, for example the presence on the Peer device of specific hardware sensors or actuators must be taken into consideration by the controller as first-priority deployment constraint. An Operator that makes use of a gyroscope sensor cannot be migrated from a smart phone built with such sensor to a Web server. ii) **Whitelists or Blacklists**. Operators can be optionally associated with a list of known Peers that must (or must not) be used to run them. The list can be specified using IP address ranges and should be large enough so that if one Peer fails another can still be found to replace it. iii) **Battery**, whenever a Peer has battery shortage, the controller should be able to migrate the Operator(s) running on such Peer in order not to completely drain out the Peer battery. At the same time, a migration operation should not be performed targeting a Peer with a low battery level. iv) **CPU**. The current CPU utilization of the Peer must leave room to deploy another operator. Since JavaScript is a single-threaded language, we use the number of CPU cores as an upper bound on the level of parallelism that a Peer can deliver.

These constraints are used to select a set of candidate Peers, which are then ranked according to additional criteria, whose purpose is to ensure that the end-to-end latency of the stream is reduced, while minimizing the overall resource utilization. This is important to reduce the cost of the resources allocated to each Topology, while maintaining a good level of performance.

4.2   Ranking Function

The following metrics are taken into account by the controller for selecting the most suitable Peer for each Operator: i) **Energy** consumption can be optimized, for example when the computation of an Operator is too taxing on a battery-dependent Peer. In this case a migration may occur, moving the heavy computation from a mobile device to a desktop or Web server machine. Thus, the priority is given to fully charged mobile Peers or to fixed Peers without battery. ii) **Parallelism** can be achieved by cloning a bottleneck Operator. For example, a migration or a cloning operation may be expected if the computation is very CPU-intensive and the Peer hosting the Operator not only has its CPU full but it is also the Topology bottleneck. Thus, higher priority is given to Peers with larger CPUs and higher CPU availability. iii) **Deployment Cost Minimization** by prioritizing Web browsers instead of making use of Web servers, WLS tries to avoid incurring in additional variable costs given by the utilization of pay-per-use Cloud resources.

These metrics can be accessed on a Web server through the Node.js APIs, while on the Web browser we can again use the HTML5 APIs to gather the battery levels and a rough estimate of the processing power. The ranking function is evaluated by the controller at Topology initialization to find the best deployment configuration of the Operators, and while the Topology is running to improve its performance and deal with variations in the set of known/available Peers. For each Peer, the function uses the previously described constraints and metrics to

---

Algorithm 1. Peer Ranking for Topology Initialization

Data: Known Peers, Topology
Result: Peers Ranking
$P \leftarrow$ Known Peers ;
foreach *Operator o in Topology* **do**
    foreach *Peer p in P* **do**
        if !*compatible*(o, p) **then**
            $P \leftarrow P \setminus p$;
        end
    end
end
if $P = \emptyset$ **then**
    return cannot deploy Topology
else
    foreach *Peer p in P* **do**
        Poll p for its current metrics $M_i(p)$;
        Compute ranking function $r(p)$ according to Eq. (1);
    end
    return Sorted list of available Peers P according to their rank $r(p)$
end

---

compute a value that describes how much a Peer is suited to host an Operator. The function is defined as follows:

$$r(p) = \sum_{i=1}^{n} \alpha_i M_i(p) \tag{1}$$

where α_i is the weight representing the importance of the $i - th$ metric. M_i is the current value of the metric obtained from the Peer p. The result gives an estimate of the utility of the Peer p to run one or more Operators on it. In case the utility turns out to be negative, the semantics implies that the Peer p should no longer be used to run any operator. Whenever a Topology has to start, the controller polls the known Peers and executes the procedure shown in Algorithm 1.

The controller first determines which of the known Peers are compatible with the Operators of the topology. Then it polls each Peer for its metrics. Once received, the Ranking is computed and stored for further use. The Peers are then ordered from the most suited to the least suited, then for each Operator to be deployed the controller iterates the list top to bottom it decides which Peer will host it. This deploys the whole Topology and when all Peers confirm that the Operators have been deployed the flow of the data stream begins. As the Topology is up and running, the ranking function is periodically re-evaluated by the controller to check the status of the execution and adapt the topology to changes in workload (reflected by changed CPU utilization of the Peers) and changes in the available Peers (and thus deal with disconnections), as well as

changes to the Topology structure itself (e.g., when new operators are removed from or added to it).

4.3   Disconnection Recovery

Whenever a Peer abruptly disconnects (i.e., a Web browser crashes), the controller notices the channel closing (through the JavaScript error event handling) and starts executing a recovery procedure in order to restore the Topology by restarting the lost Operators on other available Peers. The recovery is similar to the migration decision algorithm: the controller restarts the lost Operators on Peers satisfying the deployment constraints and with the highest ranking possible. It may be possible that the channel closing was caused by temporary network failures. In this case, if the Peer assumed lost comes back up, the controller brings it back to a clean state by stopping all Operators still running on it. The Peer may be later assigned other Operators if needed.

5   Preliminary Evaluation

The goal of the evaluation is to observe the differences between a deployment strategy to address bottlenecks using a random ranking function (random controller) as a baseline with respect to a more accurate deployment decision making approach, based on the devised ranking function defined in Eq. (1) (ranking controller). As additional control, we also performed experiments with a reverse ranking controller ($Ranking^{-1}$), which makes the worst possible decisions.

The Topology we use to evaluate the ranking controller consists of three Operators: the first takes as input a stream of tweets, the second encrypts them using triple DES and the third stores the encrypted result on a server. Given the unpredictable dynamics of the Twitter Firehose API we decided to sample 100000 tweets and use them as benchmark workload. The size of the messages exchanged along the stream is thus less than 1Kb.

For this experiment we used a Samsung Galaxy S4, one Mac Book Pro i7 with 16GB RAM running OSX, a Windows 7 machine 2.9GHz quadcore and three iPads 3 WiFi for the encryption Operator, while two servers for the producer and the consumer Operators (the producer with twenty-four Intel Xeon 2GHz cores and the consumer with four Intel Core 2 Quad 3GHz cores, both running Ubuntu 12.04 with Node.JS version 0.10.15.). All the resources using a Web browser are running Google Chrome version 40.

Figure 1 shows the resource usage of the experiments, comparing the number of processes and Peers used in the experiment. We see that in the first random approach, the deployment of the CPU-intensive Operator is on an iPad (single process), which then gets cloned on the MacBook, the best machine, resulting in a very low median latency. The other runs show a mix of deployment scenarios resulting in a not-so-optimal median throughput and with higher resource usage (two or more Peers). The ranking controller is able to detect the best machine where to run the CPU-intensive Operator and thus not only keeps the latency
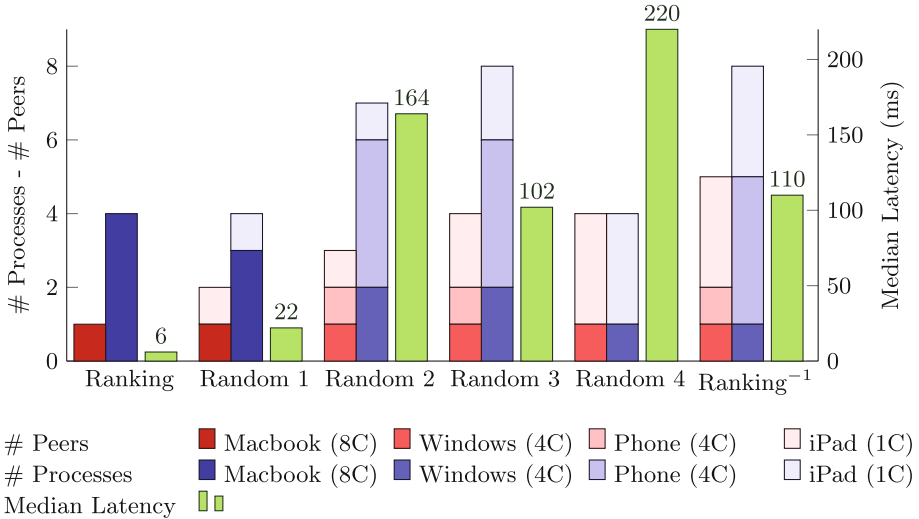
Fig. 1. Liquid Deployment and Bottleneck Handling: Comparison of Random vs. Ranked resource allocations and their median end-to-end latency

very low, but also uses a smaller amount of resources. The reverse ranking controller instead uses the iPads, the Phone and the Windows machine (in this order) to clone the Operator, resulting in a very high resource usage.

6 Conclusion and Future Work

In this paper we presented a control infrastructure for distributed stream processing across Web-enabled devices. The controller extends the Web Liquid Streams (WLS) framework for running stream processing topologies composed of JavaScript operators across Web servers and Web browsers. The control infrastructure is able to choose a suitable initial deployment on the available resources connected to WLS. While the streaming topology is running, it monitors the performance and is able to migrate or clone Operators across machines to increase parallelism as well as to prevent and react to faults given by battery shortages and disconnections. The controller is able to do so by using a ranking function which orders the Peers that are compatible with each Operator for choosing the most appropriate Peer for a deployment or migration. The initial evaluation presented in the paper shows how the controller reflects the best possible deployment given a set of resources, resulting in a balanced message throughput along the topology graph, and reduced end-to-end stream latency.

We are currently performing an extensive performance evaluation covering additional features of the WLS framework (e.g., more fine-grained CPU performance characterization, support of stateful operators, operator consolidation). Moreover, we would like to improve the current allocation approach, which is

extremely simple and greedy. It does not consider characteristics of the Operators and of the data, which may affect computation time and Peer load.

References

 1. Casteleyn, S., et al.: Ten years of rich internet applications: A systematic mapping study, and beyond. ACM Trans. Web **8**(3), 18:1–18:46 (2014)
 2. Tilkov, S., et al.: Node.js: Using javascript to build high-performance network programs. IEEE Internet Computing **14**(6), 80–83 (2010)
 3. Babazadeh, M., Pautasso, C.: A RESTful API for controlling dynamic streaming topologies. In: Proc. of WWW Companion, Seoul, Korea, April 2014
 4. Hirzel, M., et al.: A catalog of stream processing optimizations. ACM Comput. Surv. **46**(4), 46:1–46:34 (2014)
 5. Babazadeh, M., et al.: The stream software connector design space: frameworks and languages for distributed stream processing. In: Software Architecture (WICSA). IEEE/IFIP (2014)
 6. Apache: Storm, distributed and fault-tolerant realtime computation (2011)
 7. Zaharia, M., et al.: Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: Proc. of USENIX HotCloud, pp. 10–16 (2012)
 8. Akidau, T., et al.: Millwheel: Fault-tolerant stream processing at internet scale. In: Proc. VLDB Endow., pp. 734–746 (2013)
 9. Kumar, K., Lu, Y.H.: Cloud computing for mobile users: Can offloading computation save energy? Computer **43**(4), 51–56 (2010)
10. Cuervo, E., et al.: MAUI: making smartphones last longer with code offload. In: Proc. of MobiSys, pp. 49–62. ACM (2010)
11. Kosta, S., et al.: Thinkair: dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: INFOCOM, 2012 Proceedings IEEE, pp. 945–953 (2012)
12. Aniello, L., Baldoni, R., Querzoni, L.: Adaptive online scheduling in storm. In: Proc. of DEBS, pp. 207–218. ACM (2013)
13. Sax, M.J., et al.: Performance optimization for distributed intra-node-parallel streaming systems. In: ICDE Workshops, pp. 62–69. IEEE Computer Society (2013)
14. Taivalsaari, A., et al.: Liquid software manifesto: the era of multiple device ownership and its implications for software architecture. In: Proc. of COMPSAC 2014, pp. 338–343. IEEE Computer Society (2014)
15. Mikkonnen, T., Systa, K., Pautasso, C.: Towards liquid web applications. In: Proc. ICWE 2015. Springer, Rotterdam (2015)
16. Bonetta, D., Pautasso, C.: An architectural style for liquid web services. In: 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011) (2011)
17. Jelasity, M., et al.: Gossip-based aggregation in large dynamic networks. ACM Trans. Comput. Syst. **23**(3), 219–252 (2005)
18. Fuggetta, A., et al.: Understanding code mobility. IEEE Trans. Softw. Eng. **24**(5), 342–361 (1998)