HAL open science

Decentralized Stream Processing Over Web-Enabled Devices

Masiar Babazadeh, Andrea Gallidabino, Cesare Pautasso

To cite this version:

Masiar Babazadeh, Andrea Gallidabino, Cesare Pautasso. Decentralized Stream Processing Over Web-Enabled Devices. 4th European Conference on Service-Oriented and Cloud Computing (ESOCC), Sep 2015, Taormina, Italy. pp.3-18, 10.1007/978-3-319-24072-5_1 . hal-01757569

HAL Id: hal-01757569

https://inria.hal.science/hal-01757569

Submitted on 3 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Decentralized Stream Processing over Web-enabled devices

Masiar Babazadeh, Andrea Gallidabino, and Cesare Pautasso

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

Abstract. Thanks to the recent introduction of peer-to-peer communication between browsers with WebRTC, real time processing of streams can now be deployed on browsers in addition to traditional server-side execution environments. In this paper we present the Web Liquid Streams framework for building and executing stream processing topologies capable of gathering data from Web-enabled sensors and process it through JavaScript operators scattered across a peer-to-peer Cloud of computing peers. WLS offers i) support for arbitrary topologies and data streams, ii) deployment on heterogeneous Web-enabled devices, iii) transparent stream delivery across the WebRTC, WebSockets and ZeroMQ protocols, iv) stateful and stateless operators. WLS takes care of the deployment of the topology on the available resources, while users are only required to implement the operators and describe the topology graph using JSON. The structure of the topology can be dynamically adapted without stopping the stream flowing through it. We present the platform and its programming interface, showing a first evaluation of the system.

1 Introduction

As more and more sensors and smart devices are getting connected to the Internet, an interest has grown in exploring the use of the World Wide Web as platform for such devices. We have witnessed frameworks (i.e., EVRYTHNG [9]) and protocols (i.e., CoAP [13]) proposed to bridge the gap between the real-world and the Web. Some streaming applications have also started to appear [12], but there is still a lack of abstraction and flexibility for building complex stream processing pipelines [8] connecting smart devices to the Web.

To further raise the abstraction level at which stream processing and complex event processing topologies can be built across the Web of Things, in this paper we present a novel peer-to-peer streaming system that makes use of WebRTC and WebSockets to spread stream processing on both Web servers and Web browsers. The Web Liquid Streams (WLS) framework lets developers implement distributed stream processing topologies composed of operators written in JavaScript, the lingua franca of the Web. Thus, it becomes possible to deploy and run stream processing pipelines on any Web-enabled device, from small embedded microprocessors, or mobile smartphones, all the way to large virtualized Cloud computing clusters. The framework can be seen as a way to aggregate

volunteer computing resources and delivering them as a Platform as a Service (PaaS) Cloud, where stream processing applications built using JavaScript operators can be deployed and executed.

Exploiting such heterogeneous and dynamic execution environment introduces challenges in implementing and organizing the deployment of the stream processing operators. While differences in hardware platforms can be abstracted thanks to JavaScript used to implement the operators, dealing with the deployment constraints of streaming topologies, dynamic changes in the execution environment and fluctuations in the load of the streaming application can become difficult without a solid infrastructure. The WLS framework is able to guarantee the deployment of Topologies taking into account placement constraints on their operators. It also automatically deals with temporary and permanent disconnection errors by performing operator migration and recovery. Additionally, WLS also offers the possibility to alter the structure of the Topology without the need to stop it by offering an interface to add or remove streaming operators or modify their bindings at run time. This helps to adapt the topology semantics to, for example, new sensor/actuator devices that become available.

In [7] we described the RESTful API of the first version of WLS, which supported distributed stream processing only over Web server clusters, while in [6] we have shown a preliminary implementation of a controller infrastructure for the system which deals with the churn of connecting and disconnection Web browsers. In this paper we present the architecture and interface of the second version of WLS, which makes the following novel contributions: First, we show how WLS integrates heterogeneous devices and execution environments (such as Web browsers and Web servers, which may run on Web-enabled microcontrollers, mobile devices and Cloud virtual machines) in a homogeneous environment thanks to the choice of JavaScript as the operator programming language. Second, we discuss how the abstract streaming topology model is mapped to a deployment configuration taking into account multiple types of deployment constraints. Third, we demonstrate the feasibility and expressiveness of the approach through a preliminary set of experiments and case study applications.

The rest of this paper is structured as follows. Section 2 introduces the WLS framework from the developer's perspective, Section 3 is focused on its runtime operation and internal architecture. Section 4 discusses the evaluation of the system. Section 5 presents related work while Section 6 draws our conclusions and illustrates our plans for future work.

2 The WLS Stream Processing Framework

The Web Liquid Streams framework helps developers create stream processing topologies and run them across a peer-to-peer Cloud of connected devices, where they share and make use of shared resources. Developers must install the framework on their servers or microcontrollers and run a server instance of WLS, or they can connect with browsers to an existing WLS instance running to deploy operators and run them on their browsers. This Section describes the main

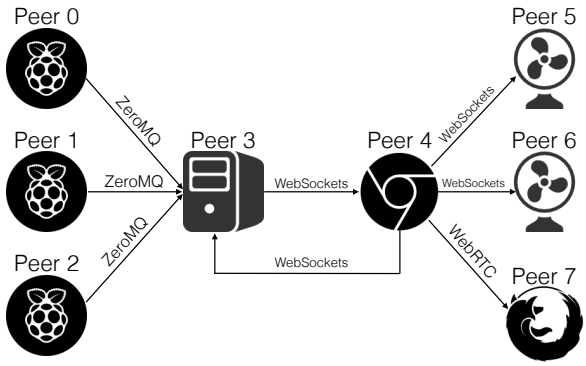


Fig. 1: Example stream deployed across Web servers and Web browsers.

abstractions provided by WLS and how they can be used to set up arbitrary Topologies of stream Operators written in JavaScript. The basic building blocks to set up and execute a Topology are Peers and Operators.

– Peers are physical hosts where the computation happens. Any Web-enabled device that can run a Web server or a Web browser can become a Peer in the Web Liquid Streams. A Peer can host more than one Operator and stateless Operators can be redundantly deployed on multiple Peers for increased scalability and reliability.

– Operators receive incoming data stream, process its elements one at a time, and forward the results downstream. Each Operator is associated to a JavaScript file, which contains the stream element processing logic.

– The Topology describes how Operators are interconnected in the data stream. It defines an arbitrary graph of operators (nodes) using data flow bindings (edges). The structure of the topology can dynamically change while the stream is running.

The example of figure 1 shows three Peers being running data producer Operators, gathering measurements through sensors. They forward the data to an Operator running on a Web server (Peer 3), which stores the temperature fluctuations over time and forwards them to another Operator running on Peer 4, which decides which actuators have to be started (Peer 5 or Peer 6) and notifies the Operator in Peer 3 about its decision. The data is also forwarded to a visualization Operator (Peer 7) which graphically shows the current status of the running actuators.

Communication links use different protocols (WebSockets, ZeroMQ, WebRTC) which are used to transfer the data stream elements. The developer of the Operators does not need to worry about the actual protocol. Based on the Topology description, it is the Web Liquid Streams runtime's task to physically deploy it on the available resources and take care of abstracting the complexity and heterogeneity of the communication channels.

2.1 System Model

We consider our system model to be composed by a large set of networked Peers owned by users of the system. Each Peer can have a different nature: from big Web servers with a lot of storage and RAM, to Web browsers running on personal computers or smartphones, down to microcontrollers and smart devices. Like Web servers are used to host one or more Web applications, Peers are responsible for their topologies, which can be deployed to be managed on different Peers.

Users of the system share their own resources (i.e., CPU, memory, sensors) in a cooperative way by either joining a Topology (i.e., connecting to a Web address) or connecting to the system from a Web server running on a smart device. WLS takes care of handling the churn of Peers by informing other Peers of new arrivals [10]. In the same way, if a Peer has to leave the network, it will notify all the Peers it knows. If a Topology is making use of the Peer about to leave, the operators running on it are automatically migrated on another available Peer.

Users of the system can take advantage of shared resources by deploying and executing their Topologies on the set of available Peers. By uploading the description of the Topology or manually creating it through the user interface (both command line and graphical interactive monitoring tool), users can start a data stream Topology execution on the shared resources. The system checks if the request to run a Topology can be satisfied with the known resources. If that is the case, it will allocate Operators on the available Peers. More than one Topology can coexist within the peer-to-peer Cloud at the same time, even at the Peer level: a Peer may host different Operators that are part of different Topologies. These slices are dynamic as they can grow or shrink depending on user needs without the need to stop the data stream. Topologies may also be modified at runtime by, for example, adding/removing one or more Operators, resulting in a structural change of semantics of the Topology itself, which will thus affect its end result.

Operators are implemented and managed by the users themselves, thus no QoS guarantee is provided. Application failures resulting from Operators crashing during the execution of the Topology should be handled by the users running the Topology. What WLS ensures is cohesion among the heterogeneous set of resources that become a stream processing platform. As a liquid adapts its shape to the one of its container, the WLS adapts the stream computation to the pool of available resources. When the stream rate or in general the stream resource demand increases, new resources are allocated. These will be de-allocated and consolidated once the stream resource demand decreases.

2.2 Topology Description

The topology structure can be built on the fly by adding or removing Operators to an already executing stream. For convenience, it is possible to capture the

configuration of a topology and represent it using JSON. The Topology description includes information about all Operators and how they are bound together. Operators can be associated with optional deployment constraints, which are used by the runtime to select a suitable Peer for running them.

Topologies must have an `id` that identifies them, a list of `operators` and a list of `bindings`. The Operators must have an `id` attribute and a link with the URL of the corresponding implementation `script`, and an optional `deploy` field which imposes deployment constraints. For example, it can specify a Peer where the Operator has to be deployed or a list of sensors that a Peer must have in order to host the Operator. If the `deploy` field is not specified, the Operator can run in any available Peer.

The list of bindings instead describes how operators are connected and which sending algorithm (Round Robin vs. Broadcast) is used when multiple operators are bound to the same upstream producer. Round Robin is used for sharding the stream across multiple operators, while Broadcast is intended to be used with multiple consumer operators that receive their copy of the stream.

2.3 Operator Script API

Operators are written in plain JavaScript and define the data processing logic. In the following example we demonstrate how to use the WLS API to write scripts which are then executed by the Operators in a simple home automation system application. The streaming application is composed by four Operators: the first one runs on microcontrollers that have access to a temperature sensor, the second one runs on servers, the third one Web browsers, while the fourth one runs on actuators that can modify the temperature in the house.

```
var k = require('wls.js');
setInterval(function(){
    getTemperature(function(temp, sensor_id){
        k.send({
        "temperature" : temp,
        "id"          : sensor_id,
        "timestamp"   : new Date()
        });
    });
}, 1000);
```

Listing 1.1: Example producer script

Listing 1.1 shows a producer script that forwards every second temperature sensor readings from the `getTemperature` function. The function reads the last measured temperature from the sensor API. The callback of the function wraps the data received into an object carrying the current timestamp and the sensor identifier and forwards it downstream using the `k.send()` function.

```
var k = require('wls.js');

//initialize operator state
```

```
k.createOperator(function(temp_data) {
    //store the received data: temperature, id, timestamp
    k.db_store("temperatures", temp_data);

    //get the last 1000 temperatures stored for that sensor id
    var average_temperature = avg( k.db_get("temperatures",
        temp_data.id, 1000) );

    //forwards the average
    k.send({
        "average" : average_temperature,
        "room_id" : temp_data.id
    });

}).start();
```

Listing 1.2: Example filter script

Listing 1.2 shows what the Web server does upon receiving the data from the sensors. First, it stores the observed temperature, computes an average over the last seen data for the room and forwards the average and the room id to the following Operator.

```
var k = require('wls.js');

k.createOperator(function(variation) {
    //get the temperature setpoint for a given room
    var setpoint = k.db_get("setpoint", temp_data.id);
    var threshold = k.db_get("threshold", temp_data.id);

    //if the temperature has to be changed, forward commands
        downstream
    if(abs(variation.average - setpoint) > threshold){
        k.send({
        "start"   : true,
        "room_id" : temp_data.id
        });
    }
    else //[...]
}).start();
```

Listing 1.3: Decision making Operator

Listing 1.3 shows a simple decision making Operator that receives objects containing an average of the last measured temperatures for a given room id. The Operator compares the received average to the setpoint and forwards a message to the actuator, which will either start or stop to reach the desired temperature.

The example can be further extended by adding sensors that notify human presence and describing new setpoints for rooms with or without people at which time of the day or night. Moreover, operators featuring machine learning could be

used to detect and predict inhabitants behavior patterns (e.g., heat the bathroom before a shower event).

There is no constraint on the data structure of the messages exchanged along a Topology, as long as they can be serialized into JSON strings. The callback function and the `send` function handle deserialization and serialization automatically, allowing the developer of the Operator script to work directly with JavaScript objects.

3 The WLS Runtime: Usage and Architecture

3.1 Starting and Handling a Topology

Before starting a Topology, the user has to upload the Topology description file and the corresponding Operator scripts to the runtime using its API. The Web Liquid Streams framework runtime is able to spread the Operators across the available Peers following the instructions of the JSON topology description file. From the command line interface, it is sufficient to type `exec topology.js` where `topology.js` is the Topology description file. This will deploy and initialize each Operator and start the flow of the data stream.

A Topology may also be built on the fly by running operators with the command `run script.js peer_id` that runs a script on an appointed Peer (if no Peer is specified, it will be run on the most suited one [6]). To bind the Operators it is sufficient to write `bind op_from op_to` where `op_from` and `op_to` are the IDs of the two Operators to be bound.

Operations that can be executed on the Topology through the command line interface include stopping an Operator (`stop op_id`), unbind two Operators (`unbind op_from op_to`) and migrate an Operator on another Peer (`migrate op_id peer_id`).

3.2 Runtime Architecture

The WLS architecture presents two main components: one runs on servers (i.e., smart devices), while the other runs on Web browsers (i.e., smart mobile phones).

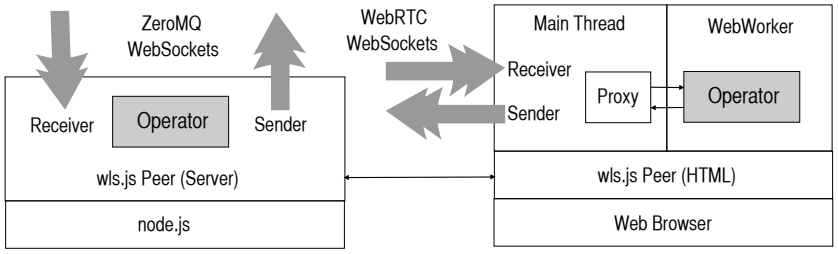


Fig. 2: Web Liquid Streams Runtime Architecture

Node.JS Peer Architecture Operators on the Node.JS Peer architecture spawn Node.JS processes to handle the incoming work. This process helps parallelizing the execution of the Operator's work, by spreading it on child processes. These child process are provided with an address and a port, thus they are directly connected to each other through ZeroMQ sockets according to the Topology bindings. The Operator takes care of adding or removing child processes in order to solve bottlenecks or free resources when possible. If the Peer hosting the Operator reaches a full CPU usage, the Peer will find another one to run an extra parallel instance of the Operator, thus solving the bottleneck.

Figure 2 (left) describes the Node.JS Peer. The grey arrows represent incoming and outgoing data streams. The Peer may host more than one Operator, for which multiple parallel instances can be started as needed. Hosted Operators may belong to different Topologies. Each parallel operator runs in its independent Node.JS process and is directly connected through the appropriate stream channel to the upstream and downstream operators. For what concerns communication to or from a Web browser, we use an adapter that can transparently perform the WebSocket-ZeroMQ (and vice-versa) protocol conversion.

Browser Peer Architecture The architecture of the client-side WLS has been implemented using plain JavaScript with WebSockets and WebRTC. The Peer and Operator code had to be adapted to run in Web browsers. To run an Operator, a Web browser has to connect to a specific path (that is, performs a `GET` request) in a Web server which is running an instance of WLS. The Web server Peer updates the list of known Peers and updates the known Peers in the network, making the Web browser a new available Peer in the peer-to-peer Cloud. The path to which the Web browser has to connect is defined on the Topology description file. This path is also associated with a given Operator, thus depending on the connection path, the Web browser will execute different Operators. The Web browser can also connect to a generic Operator page, which accepts idle Web browser Peers. During the deployment or when more computational power is needed, a streaming application can integrate those idle Peers.

The Operators in the Web browser delegate the actual execution of the script to a dedicated Web Worker thread through a proxy component. The proxy component dispatches the incoming stream elements, the Web Worker threads execute the function to process it, and send the result back to the proxy which is in charge of forwarding it downstream. The number of active Web Workers can increase or decrease depending on the load fluctuations of the data stream. Figure 2 (right) graphically shows the process including the receiver and the sender components, which are WebRTC sockets that handle communication.

Some Web browsers do not implement WebRTC yet, thus they would not be able to run an instance of the WLS Operator on them. We implemented a fallback system which is able to switch protocol to WebSockets and make the communication pass through the Web server in order to still being able to use the connected Web browser.

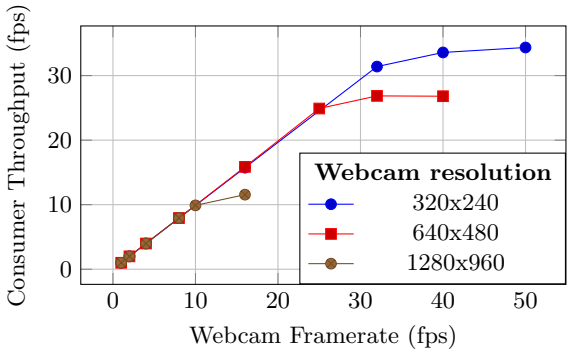


Fig. 3: Webcam resolution and framerate

4 Evaluation

We executed many experiments with WLS with different use case scenarios and deployment configurations. The current status of the implementation supports Google Chrome and Mozilla Firefox as Web browsers and Raspberry Pi as microcontroller. We also tested a deployment on Arduino through Noduino. A more lightweight version for the Tessel.IO microcontroller is being developed as we are writing this paper. In this section, we first present the results of an experiment involving Web browsers, sensors and servers in two different deployment with a variable workload scenario. The experiment shows the throughput achieved by the Topology in terms of messages per second. Then, we compare the performance of a Topology when executed on a centralized deployment on a multi-core server vs. a decentralized deployment on Web Browsers running on laptops.

4.1 Variable Workload Use Case Scenario

In this use case scenario we show how the system performs when the data throughput of the stream changes. In the first example, by increasing the throughput at the producer Operator, we increase the computational effort on the Topology and show how the system performs by parallelising on a set of finite resources. On the second example, we increase and decrease the throughput of the produces to stress the Topology constantly.

The first experiment features a linear Topology composed by three Operators. The first Operator has a deployment constraint and has to be run on a machine with a webcam. It forwards the webcam feed to a filter Operator, which applies a negative filter on the video feed and forwards the result to the final Operator which has a Web browser deployment constraint and shows the aggregated feeds of the Webcams. For simplicity, we run the Topology on three machines only, one for each Operator. The first Operator runs on a Raspberry Pi with a webcam connected. The second Operator runs on MacBook Pros i7 (2.3GHz, 4 cores) with 16GB RAM on Google Chrome (OSX) version 37, while the last one on an iPhone 5S running Chrome (iOS) version 40. Data passes through WiFi with 30mb/s

of maximum bandwidth. The WiFi is public, so external interference may be present. To increase the effort on the Topology, we increase the framerate of the webcam feed, thus increasing the workload on the filter Operator. We performed this evaluation with three different Webcam resolutions: 320x240, 640x480 and 1280x960 pixels.

The results (Figure 3) show that for a small webcam feed (320x240) we can reach a throughput of around 35 to 40 frames per seconds. By further increasing the frame rate, the system saturates the filter Operator (that is, Web Workers spawned to parallelise the execution saturated the CPU of the MacBook Pro). By doubling the resolution of the images, as expected, we see that the maximum performance that can be reached degrades in both cases (640x480 and 1280x960). It is important to notice that the parallelisation is executed only on the filter Operator, the system considers the Raspberry Pi and the smartphone too thin to be able to sustain the filter execution as well.

In the second experiment, the Topology again makes use CPU-intensive Operators that can be parallelized to process an incoming data stream with a variable data rate. This will require to use more or less of the available computing resources, depending on the actual workload. More concretely, the Topology takes as input a stream of tweets (producer), encrypts them using triple DES, and stores the encrypted result on a server (consumer). The deployment is as follows: the producer runs on the MacBook Pro i7, running the server (Node.JS) version of WLS, the encryption operator runs on a single server with twenty-four Intel Xeon 2 GHz cores running Ubuntu 12.04 with Node.JS version 0.10.15, while the consumer runs on a machine with four Intel Core 2 Quad 3GHz processors, running the same versions of Ubuntu and Node.JS.

We focused on two different workloads: the slow workload sending 40 messages per second and the fast workload sending 500 messages per second. These two workloads are alternated in a transition frequency passing from the slow workload to the fast workload every 5000 messages.We expect to see the Topology throughput to follow the fluctuating input rate of the tweets by parallelizing the work on the big machine where the bottleneck is (encryption operator).

Figure 4 shows the topology throughput of the experiment.We can see (top graph) that the fluctuating input rate (Producer) is well sustained by the system, which adapts the operator deployment configuration to deal with the increasing rate of messages. We can see in the bottom graph, how WLS parallelised the execution by allocating more Node.JS processes on the machine and de-allocating them as the load decreased.

4.2 Centralized vs. Peer-to-Peer

In this evaluation we show two different deployments of the same Topology, one on a single, big machine, and one distributed on different smaller machines (Web browsers). This evaluation shows that even deploying a Topology on a very powerful machine, by using enough weak machines we can achieve a better performance. This claim turns out to be useful when trying to minimize the

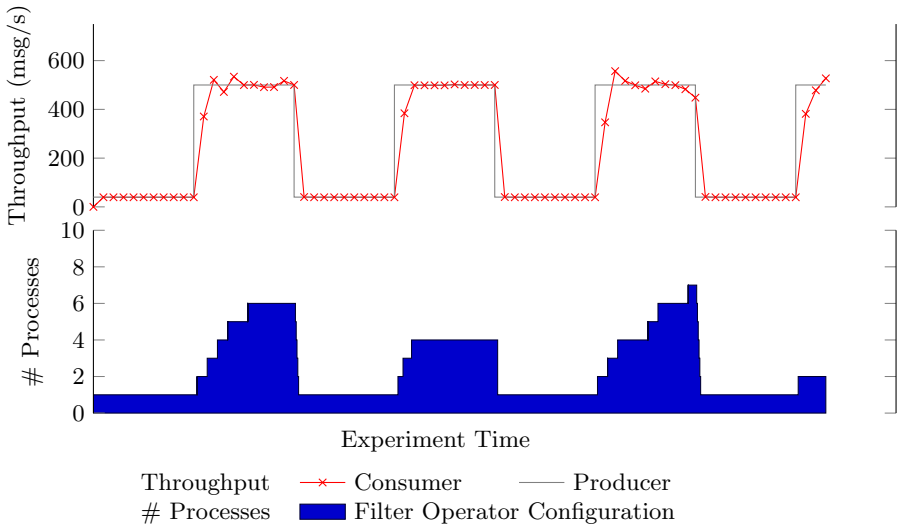


Fig. 4: Throughput and Topology deployment configuration dynamics during the workload transition experiment.

costs of deployment over Cloud servers (lower CPU usage means lower costs) in favour of a sub-Cloud of volunteer machines.

In this experiment we assess the overall throughput of the stream, i.e., the time taken to exchange a fixed number of stream items sent through the same Topology, deployed in two configurations. We have chosen to analyse the simplest Topology with a toy example in order to reduce complexity and obtain repeatable results about the deployment differences of a single Operator.

The Topology is a very simple pipeline composed by three Operators. The first Producer Operator generates a finite list of numbers which are sent one at a time downstream with a fixed data rate of 10 messages per second. The second Operator (CPU-bound filter) computes the number of prime numbers between 0 and the received number, then forwards the result to the third Consumer Operator which stores the result together with some performance metrics. The amount of data exchanged along the stream is minimal (one integer) thus reducing the impact of the network communication. The number sent by the Producer determines the time taken by the filter in computing the solution, a CPU-intensive operation. The stream begins with a relatively small number (easy work for the filter) and then switches it progressively to a bigger number (hard work for the filter). This will force the elastic parallelisation of the execution of the filter Operator up to the limit of the server machine or the peer-to-peer Cloud of available machines. We keep track of the time taken to complete the processing of the entire stream of messages as well as of the delay incurred when processing each message through the entire pipeline.

Figure 5 shows the two deployment approaches compared in this evaluation. The centralized approach is deployed on a single server with twenty-four Intel Xeon 2 GHz cores running Ubuntu 12.04 with Node.JS version 0.10.15. The

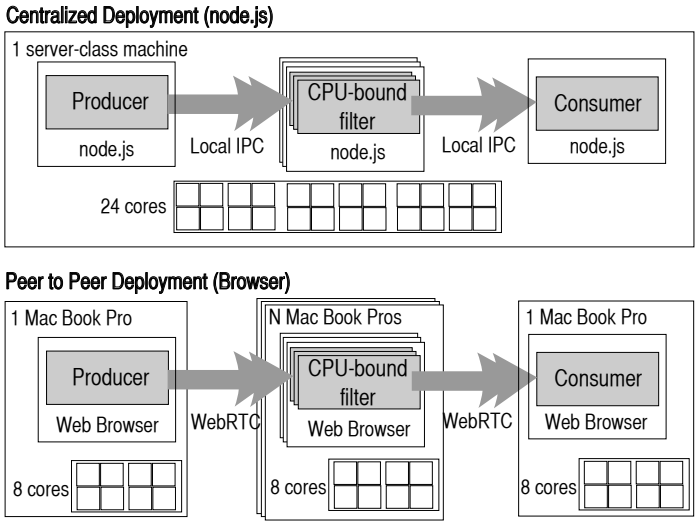
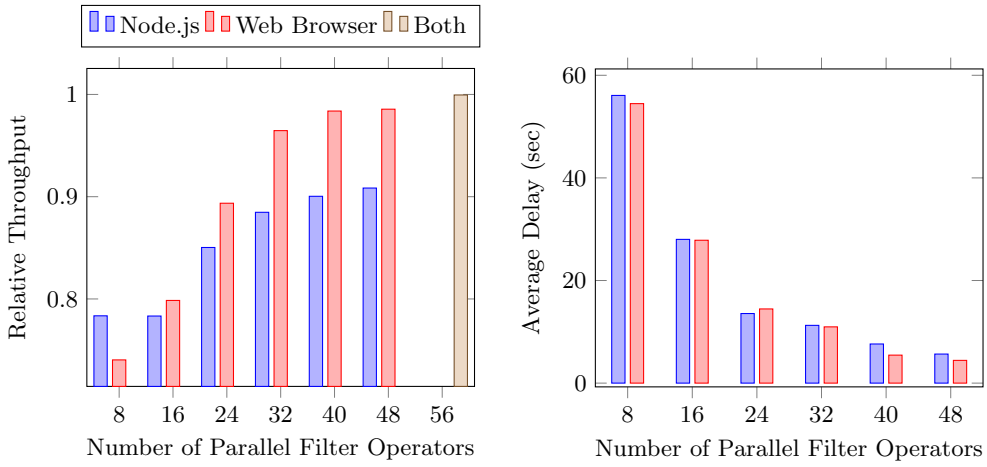


Fig. 5: Centralized vs. Peer-to-Peer Deployment Configurations

peer-to-peer approach instead is deployed only on Web browsers: we used Google Chrome v37 on MacBook Pros i7 (2.3GHz, 8 cores) with 16GB RAM.

Figure 6b shows the average delay of 10000 messages passing through the Topology. We compare the performance of the Node.JS server-side deployment on a single server-class machine vs. the behavior of the Browser only configuration on the Mac Book Pros. We can see that by increasing the number of Peers the system can increase the number of parallel filter operators and further reduce the message processing delay. This shows that the WLS framework can effectively take advantage of additional resources as they appear to improve the performance of the pipeline, whose workload can be shared among multiple peers according to their processing power.

Figure 6a shows a comparison of the throughput of the two deployments (Web browser and Node.JS) for a different number of parallel filter operators, that are distributed on all available machines. The throughput is measured at the consumer and is shown relative to the throughput of the producer operator, to indicate how well can the Topology keep up with the incoming stream data rate. The results indicate that given the same degree of parallelism, the deployment on Web browsers obtains a slightly better performance by almost reaching the same throughput of the producer. The best result was obtained for 56 parallel filters, where the Topology was deployed on both browsers and Node.JS servers. This gives a good indication of the value of the peer-to-peer Cloud concept, whereby the WLS framework can scale out a stream processing computation using opportunistic resources provided through Web browsers as well as relying on more stable foundation of server-side resources. Given an adequate amount of resources, deployment can happen on browsers only without performance loss,



(a) Relative Throughput obtained with different deployment configurations having a larger pool of peers

(b) Average delay obtained with different deployment configurations having a larger pool of peers

thus avoiding incurring in related costs of renting the corresponding amount of Cloud computing infrastructure to run the Web servers.

5 Related Work

Since the introduction of WebRTC, many browser-to-browser streaming applications have emerged. It has become possible to videoconference without the need of a central server [14], share content as in a peer-to-peer application [15] while interacting with a Web server by the means of WebSocket [16]. Still, there is a gap between the low-level messaging abstraction provided by WebSockets, the data channel of WebRTC and what is needed to conveniently build peer-to-peer streaming applications that can be deployed on a Cloud of Web-enabled devices.

Peer-to-peer, decentralized Cloud architectures have recently been recognized as an effective alternative to centralized Cloud computing infrastructures. In [4, 5], the authors proposed the design and implementation of a general-purpose framework to support distributed applications running over a very large and unreliable set of networked computing devices. We also adopted the concepts of application suite description and slicing. Like our Topologies and their constraints, the application suite describes the constraints a subcloud must have in order to be spread across the peer-to-peer Cloud. The slicing idea is embedded in the middleware component responsible for assigning the right portion of the available Cloud to the application, taking care of satisfying its requirements without using too many unnecessary resources.

A streaming processing system for the Cloud making use of sensors is Curracurrong Cloud [12] which was developed on top of Curracurrong [11], a streaming platform for Wireless Sensor Networks. Curracurrong Cloud is designed to

be deployed in large distributed clusters hosted using Cloud computing infrastructure, while maintaining the WSN deployment offered by the core platform.

Storm [1] (2011), an open source distributed real-time computational environment originally developed by Twitter. Storm's basic building blocks are called spouts (producing a data stream) and bolts (receiving streamed elements). They are used to produce and manipulate streaming data much like the concept of Operator in WLS. They can be seen as MapReduce jobs which can theoretically run forever. Spouts and bolts are executed inside Workers, physical JVMs executing a subset of the topology. Another similar approach is Discretized Streams (D-Streams) [17] (2012), a stream programming model that provides a set of transformations which treat the stream as a series of batch computations of small time intervals (reducing the latency of the jobs as much as possible).

MillWheel [2] (2013) is a Google framework that 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. The WLS runtime works at the same level of abstraction, but targets a more diverse, volunteer computing-style set of execution resources.

6 Conclusions

Stream processing is an important technology that is found in many real-time data processing scenarios. Its importance is likely to grow with the rise of the Web of Things, making it easy to process a wide variety of sensor data streams. These are typically uploaded to the Cloud for processing and storage, while the results are then downloaded to Web browsers for visualization that need to be updated in real-time.

In this paper we presented the Web Liquid Streams (WLS) framework. Developers can use it to create Topologies of connected Operators through which a data stream passes, going from data producers to data consumers. Thanks to the widespread adoption of the lingua franca of the Web, JavaScript, WLS is able to deploy operators across many heterogeneous devices, abstracting the complexity of implementing Operators and connecting them as they are deployed on Web-enabled execution environments (Web servers and Web browsers). These range from small devices (such as Arduino and Raspberry Pi), mobile phones and tablets (which run powerful mobile Web browsers), as well as traditional desktop and server class machines (which run both Web browsers and Web servers).

In this way, on the one hand we contribute an stream-centric abstraction for Web developers, while on the other we offer the capability of running a stream processing application on almost any kind of Web-enabled device without the need of installing additional software on them and without depending on a centralized Cloud infrastructure. We showed through a preliminary evaluation how the throughput of WLS Operators deals with increasing stream rate and message size. We also demonstrated that WLS can use the most appropriate communication protocol to stream data between peers and compared the corresponding overhead.

We are currently working at improving the parallelism of Operators and integrating more types of Web-enabled smart devices in the framework. We are also studying how to optimize the decision whether Operators should be deployed on Web browsers vs. Web servers and how to take into account the network latency between different Peers to determine the best possible location for an Operator. We plan to further studying the robustness, scalability and performance of WLS by porting some stream processing benchmarks to JavaScript [3], and by using WLS to build more complex Topologies in real-world use case scenarios.

References

1. Storm, distributed and fault-tolerant realtime computation, 2011. `http://storm-project.net/`.
2. T. Akidau et al. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, Aug. 2013.
3. A. Arasu et al. Linear road: A stream data management benchmark. In *Proc. of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 480–491. VLDB Endowment, 2004.
4. O. Babaoglu et al. Design and implementation of a p2p cloud system. In *Proc. of SAC12*.
5. O. Babaoglu, M. Jelasity, A.-M. Kermarrec, A. Montresor, and M. van Steen. Managing clouds: A case for a fresh look at large unreliable dynamic networks. *SIGOPS Oper. Syst. Rev.*, 40(3):9–13, July 2006.
6. M. Babazadeh et al. Liquid stream processing across web browsers and web servers. In *Proc. of ICWE 2015*, June 2015.
7. M. Babazadeh and C. Pautasso. A RESTful API for controlling dynamic streaming topologies. In *5th International Workshop on Web APIs and RESTful Design (WS-REST 2014)*, Seoul, Korea, April 2014.
8. G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.
9. D. Guinard. *A Web of Things Application Architecture – Integrating the Real-World into the Web*. PhD thesis, ETH Zurich, Zurich, Switzerland, Aug. 2011.
10. M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, Aug. 2005.
11. V. Kakkad, S. Attar, A. E. Santosa, A. Fekete, and B. Scholz. Curracurrong: a stream programming environment for wireless sensor networks. *Softw., Pract. Exper.*, 44(2):175–199, 2014.
12. V. Kakkad et al. Curracurrong cloud: Stream processing in the cloud. In *Proc. of ICDEW*, pages 207–214, March 2014.
13. M. Kovatsch. CoAP for the web of things: from tiny resource-constrained devices to the web browser. In *Adj. Proc. of UbiComp*, pages 1495–1504, 2013.
14. P. Rodríguez et al. Advanced Videoconferencing Services Based on WebRTC. In *Proc. of ICWBC*, pages 180–184, Jul. 2012.
15. C. Vogt et al. Content-centric User Networks: WebRTC as a Path to Name-based Publishing. In *Proc. of ICNP*, Oct. 2013.
16. C. Vogt, M. J. Werner, and T. C. Schmidt. Leveraging WebRTC for P2P Content Distribution in Web Browsers. In *Proc. of ICNP*, Oct. 2013.
17. M. Zaharia et al. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proc. of USENIX HotCloud12*.