Mashup Development with Web Liquid Streams

Andrea Gallidabino, Masiar Babazadeh, and Cesare Pautasso

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

Abstract. Web services such as Twitter and Facebook provide direct access to their streaming APIs. The data generated by all of their users is forwarded in quasi-real-time to any external client requesting it: this continuous feed opens up new ways to create mashups that differ from existing data aggregation approaches, which focus on presenting with multiple widgets an integrated view of the data that is pulled from multiple sources. Streaming data flows directly into the mashup without the need to fetch it in advance, making it possible to exchange data between mashup components through streaming channels. In this challenge submission we show how streaming APIs can be integrated using a stream processing framework. Mashup components can be seen as stream operators, while the mashup can be defined by building a streaming topology. The mashup is built with Web Liquid Streams, a dynamic streaming framework that takes advantage of standard Web protocols to deploy stream topologies both on Web servers and Web browsers.

Keywords: Mashups, Streaming, Liquid Software

1 Introduction

The mashup concept and the interest in the mashup tools started to appear when more and more Web services and Web Data sources were released [1]. While mashups can be built using traditional Web development tools, languages and frameworks, specialized mashup composition tools have appeared focusing on raising the level of abstraction and thus enabling non-programmers to compose mashups [2]. Different tools can be characterized depending on the users they target, and the mashup development approach they implement [3]. A precise categorisation of the various mashup tools describes synthetically their expressive power and the type of solution they propose can be found in [4].

Mashups are in general data centric applications which gather data from many Web services or Web data sources and mix them together in a single integrated application. Data may be fetched from the Web in many different forms: static resources accessible through static URLs (e.g. JSON/XML files), resources accessible through REST APIs, or – in the case of this paper – streaming and feed APIs that forward new data to clients without the need of any new request after the initial subscription. Mashup tools make it easy to integrate one or more of those type of data.

This paper presents the rapid mashup challenge solution proposed by the Web Liquid Streams (WLS) framework [5], a stream processing runtime that helps developers deploy streaming topologies running on heterogeneous Web-enabled devices. WLS helps the users to develop logic mashups by creating JavaScript logic components. Components may interact with any of the data sources described above and components may be connected together in order to create a streaming topology representing the mashup.

2 Related Work

As witnessed during the challenge, there exists many mashup tools based on different paradigms and runtime architectures. By using the Web browser as a platform, many tools implement the logic of both the **integration** and the **presentation** directly on the Web browser. While the presentation layer suits perfectly the Web browser environment, there are some issues with the integration layer, which can not always be fully deployed on the Web browser [6]. The solution to this problem is decoupling the integration and presentation layer by shifting the development of mashup from the client-side to the server-side [7]. With Web Liquid Streams, it is possible to dynamically decide where mashup components should be deployed.

Many mashup tools take advantage of the data flow paradigm to represent how information and events flow between mashup components connected into pipelines. Tools like *FeedsAPI*[1], *Superpipes*[2], or *Yahoo Pipes*[3] use pipelines as a mechanism for developing mashups. The idea is to create a **flow** of data that goes from a multitude of sources through one or many integration layers and finally ends the flow in the presentation layer.

The pipeline approach can be easily implemented in a streaming framework [8] because all the layers of a pipelined mashup can be directly translated to a type of operator in a streaming topology: data sources used in a mashup translate to *producer* operators, the integration layers translate to *filter* operators, and the presentation layer translates to *consumer* operators. Even if streaming frameworks naturally suit the implementation of pipelines, not all streaming frameworks are suited for mashup development. Mashup development with streaming frameworks must meet two criteria: operators must be able to interact with external data sources and Web APIs and there must be a mechanism enabling visualisation of the consumer operators and their deployment as Web Widgets.

JOpera is a process-based [9] mashup composition tool that was extended in 2009 with streaming execution support to build real-time mashups of stream data sources found on the Web [10].

[1] http://www.feedsapi.com/
[2] https://github.com/superfeedr/superpipes
[3] https://pipes.yahoo.com/pipes/

Chrooma+ [11] is a streaming mashup tool that enables construction of mashups with video and audio sources. It can create composition of media streams with any HTML component.

SensorMasher [12] uses streams of data produced by sensors as data sources and builds visual compositions on a Web browser. SensorMasher publishes the data of the attached sensors as Web data sources, making it possible to integrate them with other data sources.

3 Web Liquid Streams Framework

WLS helps Web developers to create streaming topologies running across heterogeneous Web-enabled devices. Any device on which a Web browser or a node.js Web server can run, can be used to produce, process or consume a WLS stream. WLS targets programmers that are able to write JavaScript code that runs both on the server and on the client. Mashup components in WLS are called *operators* and may interact with any Web service API (both streaming, RESTful and RPC-based). An operator is the core building block of a streaming topology, it can receive data, process it and forward results downstream. By *binding* (connecting) two or more operators together it is possible to define a streaming topology.

Operators may run on Web servers or on Web browsers. In both environments they can use the same WLS API to produce and consume the data stream (see Section 3.4). Operators running on a Web Browser also have access to an extended API for rendering the data stream and visualize it on web pages (see Section 3.5). Figure 1 shows the scheme of a possible topology composed by three operators: the producer and filter run in the server-side while the producer runs on a browser. In this example the filter makes requests to an external API.

WLS abstracts away the deployment on the heterogeneous machines from the development of the topology. It keeps the mashup alive in case of failures. If a mashup component overload is detected it automatically allocates more resources to that component [13].

In this section we discuss the features offered in the WLS framework in more detail. Explanations are followed by real examples used in the demo.

3.1 Startup and Discovery of the Devices

Discovery of the devices happens by direct connection through different entry points provided by the WLS runtime. When WLS finishes the initialisation, a list of ports is prompted on the screen (Figure 2). Every port defines a specific entry point or a service published by the WLS application. Smart devices running the WLS-node script can connect to the application by connecting to the *RCP* entry point, while Web browser enabled peers can connect to the *Remote* entry point. Through the *HTTP* port it is possible to send HTTP requests.

WLS also publishes a RESTful API [14] through the *REST API ROOT* port. During the demo we show how to retrieve topology descriptions through this interface (Fig.9).

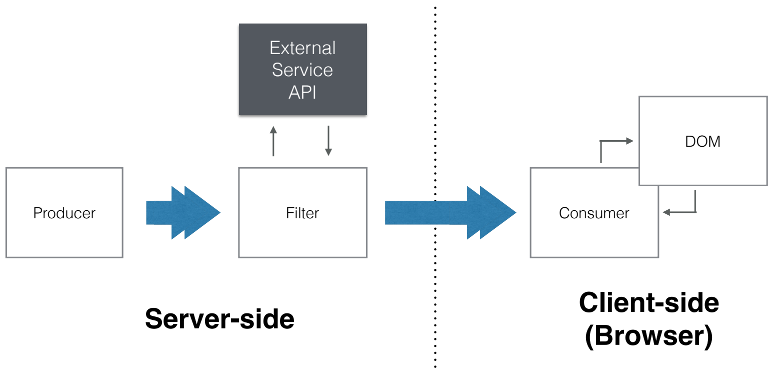
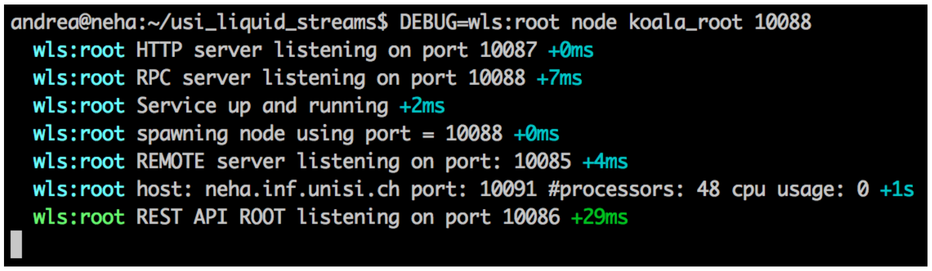


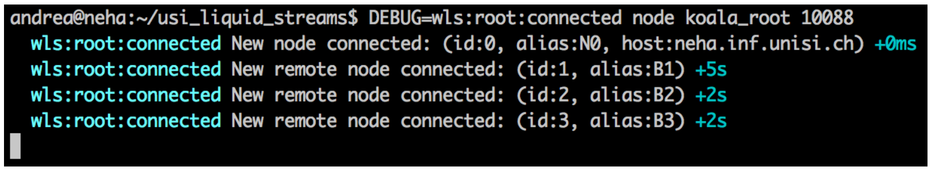
Fig. 1: Web Liquid Streams topology example



```
andrea@neha:~/usi_liquid_streams$ DEBUG=wls:root node koala_root 10088
  wls:root HTTP server listening on port 10087 +0ms
  wls:root RPC server listening on port 10088 +7ms
  wls:root Service up and running +2ms
  wls:root spawning node using port = 10088 +0ms
  wls:root REMOTE server listening on port: 10085 +4ms
  wls:root host: neha.inf.unisi.ch port: 10091 #processors: 48 cpu usage: 0 +1s
  wls:root REST API ROOT listening on port 10086 +29ms
```

Fig. 2: Web Liquid Streams startup

When devices connect to the WLS application they become *Peers* and an unique peer id (*pid*) is assigned to them. Figure 3 shows the WLS application's answer to the connection of one server peer and three remote peers connected through a Web browser.



```
andrea@neha:~/usi_liquid_streams$ DEBUG=wls:root:connected node koala_root 10088
  wls:root:connected New node connected: (id:0, alias:N0, host:neha.inf.unisi.ch) +0ms
  wls:root:connected New remote node connected: (id:1, alias:B1) +5s
  wls:root:connected New remote node connected: (id:2, alias:B2) +2s
  wls:root:connected New remote node connected: (id:3, alias:B3) +2s
```

Fig. 3: Web Liquid Streams discovery

3.2 Commands

WLS provides a list of console commands allowing live development of a mashup:

run script pid (e.g. *run producer.js 1*)
Creates a new operator on peer *pid* and loads the defined *script*. Workers inside the operator will run the loaded script. The *run* command returns a unique operator identifier (*cid*) representing the created operator.

bind cidFrom cidTo (e.g. *bind 1 2*)
Creates a one-way communication channel between two operators: from operator *cidFrom* to operator *cidTo*.

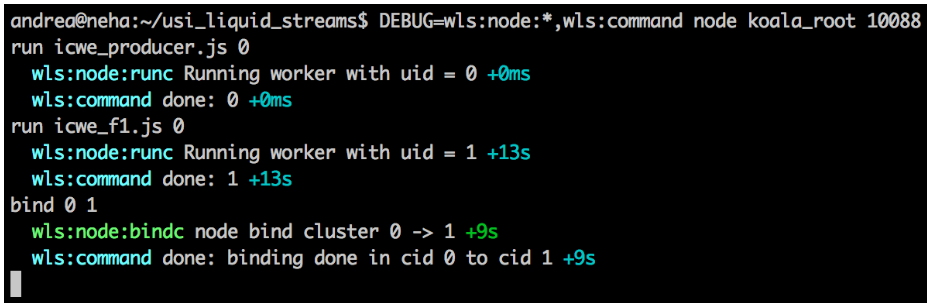
kill cid (e.g. *kill 1*)
Unloads all the workers inside the operator *cid* and kills it.

unbind cidFrom cidTo (e.g. *unbind 1 2*)
Removes the communication channel created from operator *cidFrom* to operator *cidTo*.

exec topology (e.g. *exec icwe_topology.js*)
Given the definition of a topology, automatically runs the needed **run** and **bind** commands in order to deploy the operators on the available peers.

migrate cid pid (e.g. *migrate 1 0*)
Move operator *cid* from its current peer to peer *pid*.

Figure 4 shows a real example of running and binding operators. Two operators are run with scripts *icwe_producer.js* and *icwe_f1.js* on pid 0. The run command returns two *cid*: the runtime assigns to the first operator cid 0 and to the second one cid 1. When both operators are started it is possible to bind them with the bind command.



```
andrea@neha:~/usi_liquid_streams$ DEBUG=wls:node:*,wls:command node koala_root 10088
run icwe_producer.js 0
  wls:node:runc Running worker with uid = 0 +0ms
  wls:command done: 0 +0ms
run icwe_f1.js 0
  wls:node:runc Running worker with uid = 1 +13s
  wls:command done: 1 +13s
bind 0 1
  wls:node:bindc node bind cluster 0 -> 1 +9s
  wls:command done: binding done in cid 0 to cid 1 +9s
```

Fig. 4: Running and binding example

3.3 Topology

A topology can be described with our internal DSL based on the JSON syntax[4]. The description defines both *operators* and *bindings* as follows:

operators
- **id** Identifier of the operator
- **script** Script loaded in the operator and ran by the workers
- **browser** Defines if an operator may run on the client-side, if not defined the operator exists exclusively on the server-side
  - **path** Relative path of the domain that enables direct access to the operator through a Web browser connection

[4] http://json.org/

only If *true* an operator runs exclusively on the client-side, if *false* the operator may run on both server-side and client-side

bindings

- **from** Identifier of an operator defined in the operator array
- **to** Identifier of an operator defined in the operator array
- **type** Sending algorithm such as: *round-robin* or *broadcast*

Listing 1.1 shows the implementation of a linear topology composed by three operators: the first and second operators can run only on the server-side, while the third one can run only in a browser and is accessible to the URL */map*. The first operator sends messages to the second one in a round-robin fashion, the second broadcasts messages to the third one.

Listing 1.1: Topology JSON description example: linear topology with three operators used in the first iteration of the demo

```
{
    "topology": {
        "id": "test",
        "operators": [{
                "id": "producer",
                "script": "icwe_producer.js"
            },{
                "id": "filter",
                "script": "icwe_f1.js"
            },{
                "id": "consumer",
                "script": "icwe_browser.js",
                "browser": {
                    "path" : "/map",
                    "only" : true
                }
            }
        ],
        "bindings": [{
                "from": "producer",
                "to": "filter",
                "type": "round_robin"
            },{
                "from": "filter",
                "to": "consumer",
                "type": "broadcast"
            }
        ]
    }
}
```

3.4 Script API

WLS provides developers with the following basic API:

var wls = require('wls.js')
An operator script has to import the *WLS* library. The library contains the two streaming routines needed to create the topology's streaming flow. Our framework redefines the *require* function in the remote clients in order to make server-side scripts and client-side scripts as compatible as possible without the need of any further modification. It is important to note that in the **client-side** the *require* function should be called only once in order to load the WLS library, since it always returns the *WLS* object no matter the arguments passed (it does not load any server-side node modules).

wls.createOperator(function(message){. . . })
The *createOperator* method is used to execute scripts on messages coming from upstream. It takes a **callback** function parameter which is executed every time a message is parsed by the operator. The callback function receives the *message* itself as the first argument. A script can define only a single operator.

wls.send(message)
The *send* method is used to send messages downstream to all operators bound to the sender. The message must be a serializable object. We highly recommend to send JSON parsable objects as messages.

Listing 1.2 shows the implementation of one of the scripts used in the rapid mashup challenge (Section 4). The script receives a message from upstream as an argument to the callback registered with *wls.createOperator* (lines 4-8). When processing every stream message, the script makes an external HTTP request in the *geoNamesRequest* function (lines 10-25). The answer to the request is eventually forwarded downstream through the *wls.send* function (lines 18-22).

Listing 1.2: Server script: Tweet Geolocate

```
1  var wls = require('wls.js')
2  var http = require('http')
3
4  wls.createOperator(function(msg) {
5      var tweet = msg.tweet
6      var locationName = getLocationName(tweet)
7      geoNamesRequest(locationName, tweet)
8  })
9
10 var geoNamesRequest = function(locationName, tweet){
11     var options = {...}
12
13     http.get(options, function(res) {
14         var coords = undefined
15         ...
16
```

```
        res.on('end', function () {
            wls.send({
                tweet: tweet,
                color: createRandomColor(),
                location: coords
            })
        })
    })
}
// Returns the name of a location connected to the tweet
var getLocationName = function (tweet) {...}
// Returns a random color
var createRandomColor = function () {...}
```

3.5 Extended Remote Script API

Implementation of WLS in the Web browser is slightly different from the server-side. Workers in the server-side are spawned as child-processes of the WLS run-time, while in the Web browser workers run as WebWorkers. Scripts running in a Web browser should be able to access and interact with the Document Object Model (*DOM*) of the Web page, but WebWorkers lack direct access to the DOM. The remote peers in the browsers have access to an extended set of API methods that enhance the communication between the *DOM* and the operator's script.

wls.createHTML(id, html)
The *createHTML* method adds HTML code snippets to the DOM. It takes two parameters: a unique *id* and the HTML code snippet passed as a String.

wls.createScript(id, scriptPath)
The *createScript* method adds a client-side script to the header of the associated Web page. It takes two parameters: a unique *id* and the path to the script relative to the domain name.

wls.callFunction(name, argumentsArray [, function(result){...}])
The *callFunction* method calls a function associated to the *DOM* from within the WebWorker. If the *DOM* defines a function named *name*, then it will be executed by passing the *argumentsArray* as the arguments. If the optional callback function is passed as an argument, it will be executed after the function call ends. The callback takes the returned value of the executed function as the first parameter.

wls.setDOM(selector, attribute, value)
The *setDOM* method sets a new *value* to the *attribute* of the specified *DOM* elements. The elements are specified by the *selector* parameter and are written as *jQuery selectors*[5].

wls.subscribe(id)
The *subscribe* method creates a direct communication channel from the *DOM* to the WebWorker. Once the WebWorkers subscribe, the *DOM* can create

[5] https://api.jquery.com/category/selectors/

and send messages to the WebWorkers as if the *DOM* is an operator in the topology. It can send messages through the channels with the framework function **WLS.publish(id, message)**, where *id* is the unique identifier specified in the *subscribe* call and *message* is the object forwarded by the *DOM*.

Listings 1.3, 1.5, and 1.6 show the implementation of the three remote scripts used in the rapid mashup challenge (Section 4).

In Listing 1.3 we show the script that creates markers inside the GoogleMap. The first time the script is loaded it will inject the GoogleMap HTML into the DOM by using the function *wls.createHTML* (line 7) and it will inject into the header of the Web page the script 'js/map.js' (Listing 1.4) by using the function *wls.createScript* (line 8). When a message arrives from upstream (lines 1-6) the message is processed and the worker will call the *addMarker* function which is now defined in the DOM by using the *wls.callFunction* method.

In Listing 1.5 we show the script that visualises information associated to the markers on the GoogleMap. The first time the script is loaded it registers a new subscriber by calling the *wls.subscribe* method (line 9). Whenever in the DOM the method *WLS.publish('markermouseover', msg)* is called (Listing 1.4: line 8-10), a message is forwarded to the worker script, as if it had received a message from upstreams (lines 2-8). Once the message is processed, the script will modify some attributes of the Web page by calling the *wls.setDOM* method (lines 5-7).

Similarly in Listing 1.6 the script registers a subscriber called *markerclick*(line 9). Wherever the DOM calls the method *WLS.publish('markerclick', msg)* (Listing 1.4: line 4-6) a message is sent to the worker script.

Listing 1.3: Browser Script: Marker Creator

```
1 wls.createOperator(function(msg) {
2   var tweet = msg.tweet
3   var color = msg.color
4   var location = msg.location
5   wls.callFunction('addMarker',[tweet,color,location],undefined)
6 })
7 wls.createHTML('mapDiv', '<div id="map-canvas"></div>');
8 wls.createScript('mapScript', 'js/map.js');
```

Listing 1.4: js/map.js script

```
1 var addMarker = function(tweet, color, location) {
2     ...
3 
4     google.maps.event.addListener(marker,'click',function(){
5         WLS.publish( 'markerclick', {tweet:tweet, color:color,
              count:count})
6     }
7     ...
8     google.maps.event.addListener(marker,'mouseover',function(){
        WLS.publish('markermouseover', {tweet:tweet, color:color
            })
    }
    ...
}
```

Listing 1.5: Browser Script: Marker Viewer

```
var wls = require('wls.js')
wls.createOperator(function(msg) {
  var tweet = msg.tweet
  var color = msg.color
  wls.setDOM('#marker_color', 'css', "background-color", color)
  wls.setDOM('#marker_author', 'html', tweet.user.screen_name)
  wls.setDOM('#marker_tweet', 'html', tweet.text)
})
wls.subscribe('markermouseover')
wls.createHTML(...);
```

Listing 1.6: Browser Script: Marker Clicker

```
var wls = require('wls.js')
wls.createOperator(function(msg) {
  wls.send({
    tweet: msg.tweet,
    color: msg.color,
    count: msg.count
  })
})
wls.subscribe('markerclick')
```

4 Rapid Mashup Challenge

Figure 5 summarizes the final topology deployed during the demo, the description of the scripts can be found in Section 4.2. The mashup we propose in the demo mixes the following three external APIs:

Geonames
: Geonames[6] converts name Strings to a pair of latitude-longitude coordinates. Answers from the GeoNames API are in JSON format, which can be forwarded downstream without the need of any processing.

GoogleMaps
: GoogleMaps[7] adds a geographic map to a Web page, its API allows creation of markers on the map given the latitude-longitude coordinates.

[6] http://www.geonames.org/
[7] https://developers.google.com/maps/

Twitter REST API[8] and Streaming API[9]

REST: The Twitter REST API is used to retrieve all the re-tweets associated to a given tweet.

Streaming: We subscribe to the streaming feed of *The New York Times* (TNYT) Twitter account *@nytimes*. Every time The New York Times tweets a piece of news, a message is forwarded to our operator.

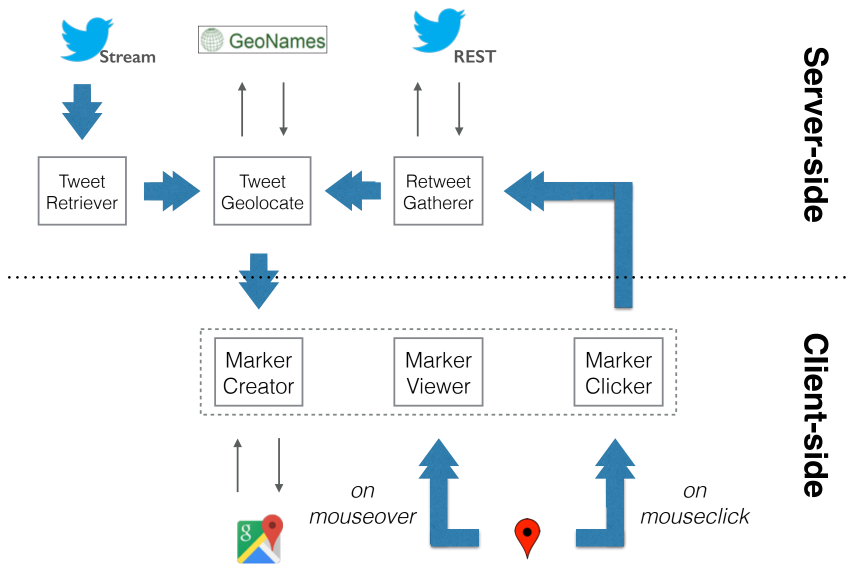


Fig. 5: Complete Stream Topology and Component Deployment

The rapid mashup challenge demo mashup marks on a map the geographical location of the news published by *The New York Times*. Moreover the mashup detects two different events associated to the marker: when the mouse is over a marker the mashup returns additional textual information about the news; when a marker is clicked it shows on the map, with smaller markers of the same color, the geographical location of all the users who retweeted it.

During the challenge we incrementally build the mashup from scratch, starting with the definition of a simple linear topology with our JSON syntax. The initial topology only shows the tweets on the GoogleMap.

After the initial solution, we expand the topology dynamically by invoking the console commands described in Section 3.2. The extended topology is now non-linear, it re-uses the *Tweet Geolocate* component, and offers the *onclick* and

[8] https://dev.twitter.com/rest/public
[9] https://dev.twitter.com/streaming/overview

mouseover functionalities. The extended mashup has been obtained by incrementally adding components to it without stopping its execution, a form of live mashup development [15].

At the end of the demo we ask the audience to connect to our Web application. Anybody that connects to the mashup with his Web browser will be able to see the GoogleMap. With the new clients connected to WLS we demonstrate the migration and the distribution of the mashup components over the set of peers contributed by the audience.

4.1 Motivation

The proposed mashup allows the demonstration of the most important features offered by WLS:

Topology definition Topologies can be created by the means of our DSL language and executed automatically.

Live mashup development Topologies can be extended at runtime and components can be added or removed while already existing mashups are running.

Reusable components Components are independent from the topology they were created for. The *Tweet Geolocate* is used both in the initial topology and the extended one with different upstreams operators.

Distributed user interface mashups The live demo shows that more than one operator can be instantiated to visualise the data. The stream topology can be deployed on different clients and therefore its results are shared among multiple users.

4.2 Scripts

The rapid mashup challenge is composed by three server-side scripts and three client-side scripts.

4.2.1 Server-side

Tweet Retriever
The operator subscribes to the feed of 'The New York Times' Twitter. When a new tweet is forwarded to the operator it is processed and trimmed of the useless data. The processed tweet is forwarded to the *Tweet Geolocate* operator.

Tweet Geolocate
The implementation of this operator can be found in Listing 1.2. The Twitter feed does not directly return the latitude-longitude coordinates of the news. This operator searches for a location name inside the tweet and sends an HTTP request to the GeoNames API. The tweet and the GeoNames answer is **broadcasted** downstream to all *Marker Creator* operators.

Re-tweet Gatherer

The operator receives a tweet and a number from upstream. It sends an HTTP request to the REST Twitter API and requests the first n re-tweets connected to the given tweet, where n is the number received from upstream. Every single retweet is then forwarded in a message to the *Tweet Geolocate* operator.

4.2.2 Client-side

Marker Creator

The implementation of this operator script can be found in Listing 1.3. This operator injects the HTML code and javascript of the GoogleMap into the Web page. Every time a tweet arrives from upstreams it calls a function defined in the DOM which adds a marker on the GoogleMap. Figure 6 shows the map on the Web page with some markers on it. Big markers are news tweeted by the TNYT, while small markers are retweets.

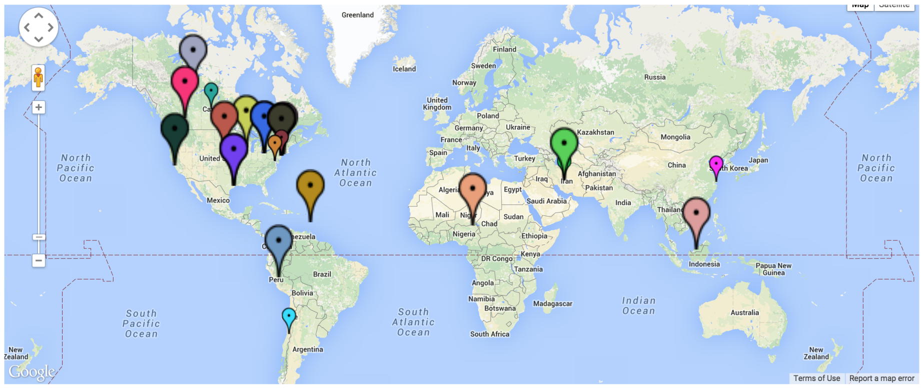
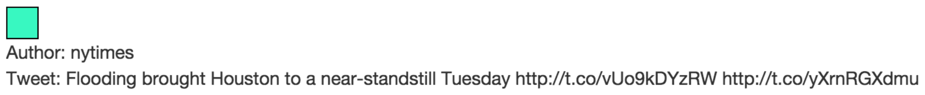


Fig. 6: Marker Creator

Marker Viewer

The implementation of this operator script can be found in Listing 1.5. This script adds an HTML tweet viewer to the Web page (Figure 7). When the *mouseover* event of a marker is fired the operator receives the tweet as a stream message and changes the page accordingly by showing the most relevant information on the Web page: marker color, author, and text of the tweet.



🟩

Author: nytimes

Tweet: Flooding brought Houston to a near-standstill Tuesday http://t.co/vUo9kDYzRW http://t.co/yXrnRGXdmu

Fig. 7: Marker Viewer

Marker Clicker

The implementation of this operator script can be found in Listing 1.6. This script adds a number picker to the Web page. When the *click* event of a marker is fired the operator receives a message with the tweet and the number selected in the picker. The operator sends a request downstream to the *Retweet Gatherer* with both information.

5 Demo

We present our plan for the rapid mashup challenge with a five-phase demo. Scripts are defined in advance and are not discussed during the challenge demonstration.

5.1 Slides Presentation

The slides presentation[10] gives the audience a general introduction to WLS. The presentation describes the concept of *operators*, *binding*, and *topology*. Moreover it presents the console commands used during the demo: *run*, *bind*, *exec*, and *migrate*. Lastly it introduces the JSON description of a topology and the possibility to dynamically change it at runtime.

5.2 Startup and Connection

We open the demo by starting the application and explaining the console log messages (see Section 3.1). In particular we not only show the audience the different ports and services, but specifically the connection of the peers. We explain that the server itself is a peer and the application assigns pid *0* to it.

Aftwerwards we open a Web browser (Chrome or Firefox) and connect to the framework through the *remote* port. The console logs the connection of a new remote peer and assigns the pid *1* to it.

At this point the set of peers managed by the application consists of two peers: a server with pid *0* and a remote peer with pid *1*.

5.3 Topology Creation and Deployment

We create the JSON description of a linear topology. The topology is created on a text editor so that people can see the description of the *operators* and *bindings*. We construct the topology iteratively allowing the audience to connect what they see with what they heard during the slides presentation.

We use an empty JSON template (Listing 1.7) as a starting point for the creation. After briefly explaining the template we add the three operators *Tweet Retriever*, *Tweet Geolocate*, and *Marker Creator*. In particular we make sure to

[10] http://www.slideshare.net/AndreaGallidabino/web-liquid-streams-mashup-challenge-icwe-2015

explain the audience that the first two operators will run on the server while the last one will run on a Web browser by defining the *browser* flag (see Section 3.3).

Lastly we add to the description two bindings connecting the three operators, making sure to explain the difference between the *round-robin* and *broadcast* sending algorithm. The final topology description can be viewed in Listing 1.1.

Listing 1.7: Topology description starting point

```
{
    "topology": {
        "id": "test",
        "operators": [
            "ADD OPERATORS HERE"
        ],
        "bindings": [
            "ADD BINDINGS HERE"
        ]
    }
}
```

After the definition of the JSON file we can execute the topology. We remind the audience about the console commands and call *exec*. The console prompts messages as if it had executed three *run* commands and two *bind* commands (see Section 3.2). Figure 8 shows the view created on the Web browser: it shows the pid *1* on the top, the GoogleMap and the cid *2* which is the same operator identifier written on the console (meaning that the GoogleMap was added to the Web page by the *Marker Creator* operator).

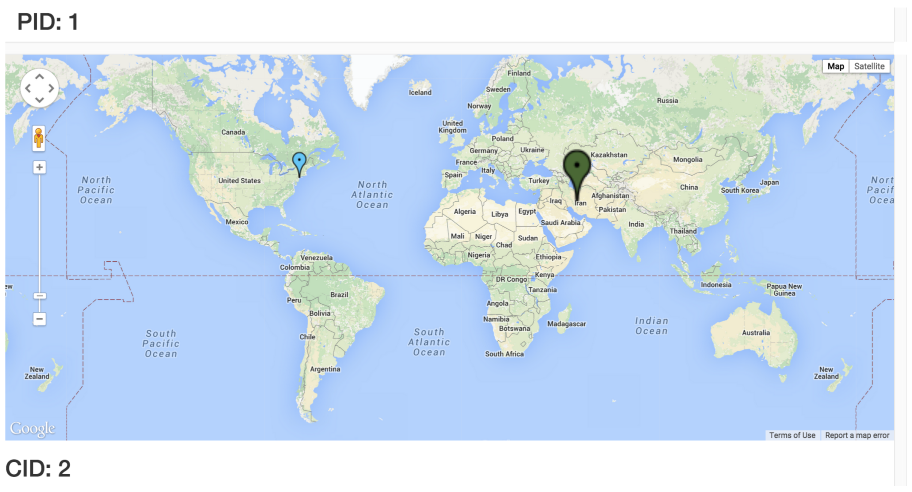
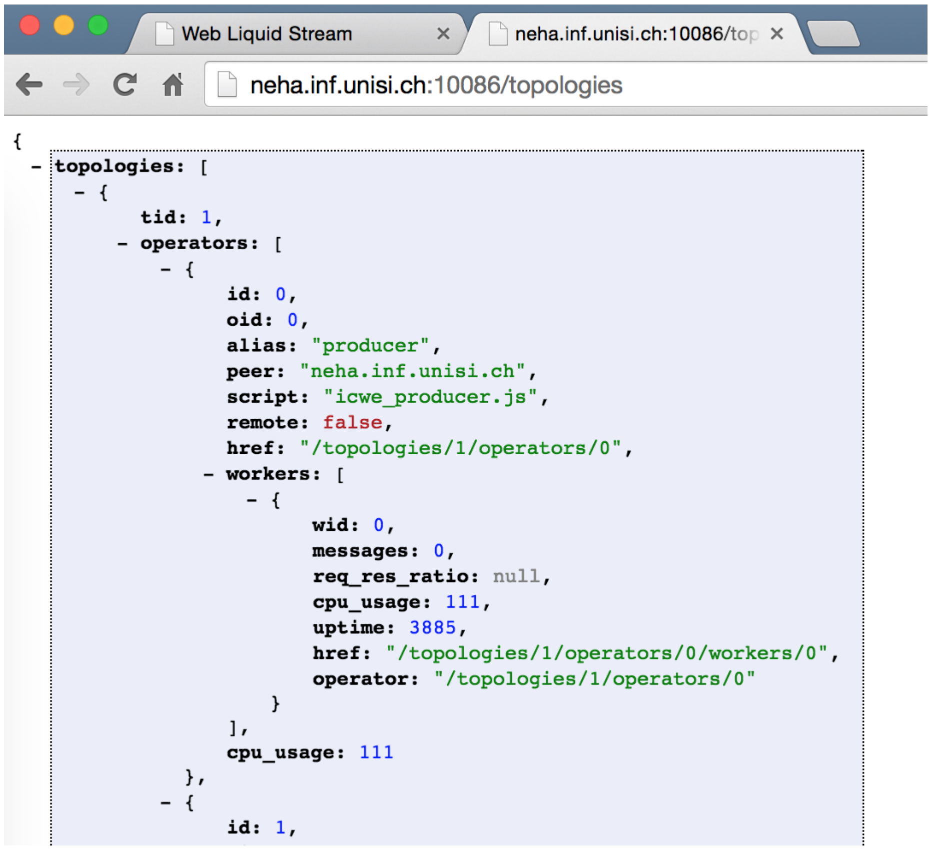


Fig. 8: First Iteration: News Stream on the Map

Then we show that by connecting to the *REST API* port (see Section 3.1), we can see the current topology ran by the application. Figure 9 shows the extended JSON description displayed on the Web browser. This description has many more fields than the one described previously because it also contains information about the runtime of the operators (such as the *cpu_usage* or *workers*).


Web Liquid Stream
neha.inf.unisi.ch:10086/top
neha.inf.unisi.ch:10086/topologies

```
{
  - topologies: [
    - {
          tid: 1,
        - operators: [
          - {
                id: 0,
                oid: 0,
                alias: "producer",
                peer: "neha.inf.unisi.ch",
                script: "icwe_producer.js",
                remote: false,
                href: "/topologies/1/operators/0",
              - workers: [
                - {
                      wid: 0,
                      messages: 0,
                      req_res_ratio: null,
                      cpu_usage: 111,
                      uptime: 3885,
                      href: "/topologies/1/operators/0/workers/0",
                      operator: "/topologies/1/operators/0"
                  }
                ],
                cpu_usage: 111
            },
          - {
                id: 1,
```

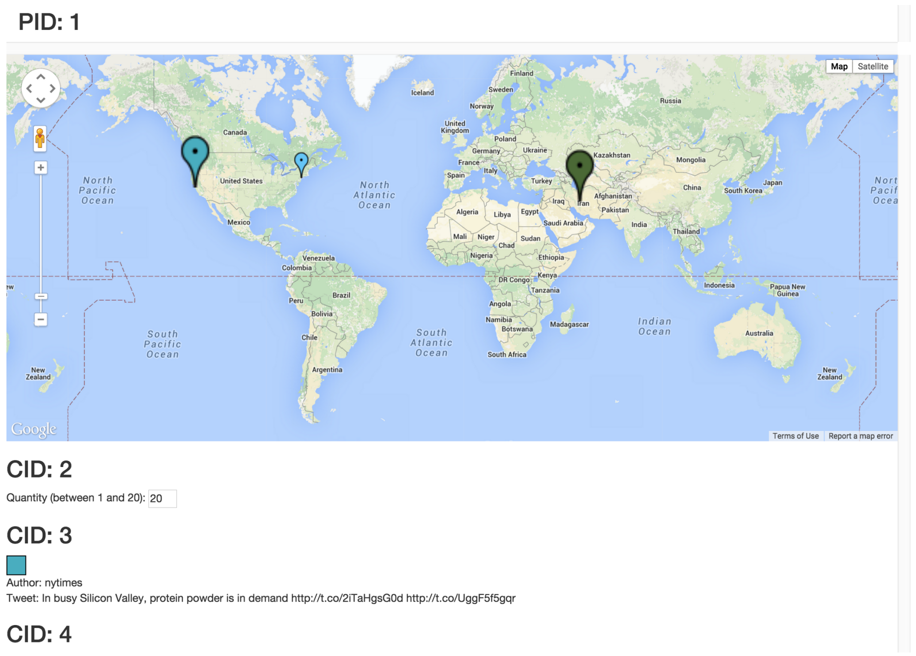
Fig. 9: Topology description retrieved from the REST API

At this point the Web browser displays the map and every time a tweet or retweet arrives to the client operator a randomly colored marker drops on the map.

5.4 Topology Development

We dynamically extend the current topology so that we construct the complete solution to the rapid mashup challenge. We use the console commands *run* for the three scripts *Marker Viewer*, *Marker Clicker*, and *Retweet Gatherer*.

Once the run commands finish their execution we show the changes on the Web browser (Figure 10). The Web page now contains some new HTML, in particular we see a number picker added by the *Marker Clicker* operator with cid *3* and the tweet viewer added by the *Marker Viewer* operator with cid *4*.



PID: 1

CID: 2

Quantity (between 1 and 20): 20

CID: 3

🟦

Author: nytimes

Tweet: In busy Silicon Valley, protein powder is in demand http://t.co/2iTaHgsG0d http://t.co/UggF5f5gqr

CID: 4

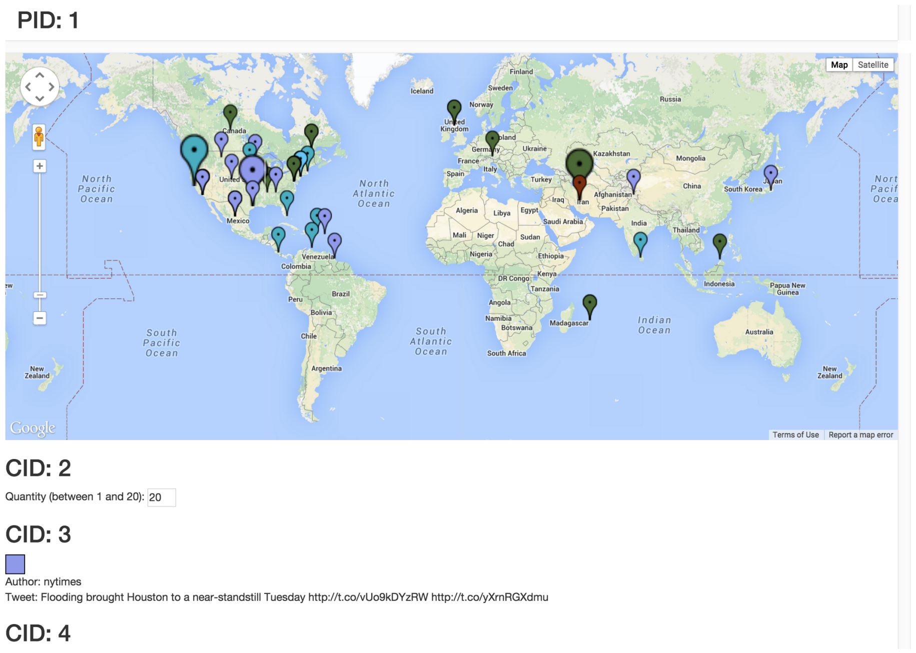
Fig. 10: Second Iteration: Marker Creator, Marker Viewer, and Marker Clicker views

We show that by triggering the *mouseover* event on a marker the *Tweet Viewer* HTML changes accordingly, while if we try to trigger the *click* event nothing happens. We explain to the audience what is missing, i.e.the fact that we still did not create the bindings to the *Retweet Gatherer*. We run on the console the two needed *bind* commands, connecting the *Retweet Gatherer* to the *Marker Clicker* and to the *Tweet Geolocate* operators.

Finally we can show the audience that by clicking on a marker in the map the retweets appear. Figure 11 shows the final outcome of the demo on the Web browser: if a tweet (big marker) is clicked, then many retweets (small markers) of the same color appear on the map.

5.5 Live Demo and Migration

We invite the audience to participate in the demo and give them the application URL. They connect to the direct address of the map operator defined in the JSON topology (Listing 1.8). Anybody connecting to the application can see


PID: 1

CID: 2

Quantity (between 1 and 20): 20

CID: 3

🟦

Author: nytimes

Tweet: Flooding brought Houston to a near-standstill Tuesday http://t.co/vUo9kDYzRW http://t.co/yXrnRGXdmu

CID: 4

Fig. 11: Fully functional rapid mashup challenge

the GoogleMap and receive the markers on their Web browser. We show that we can move the map from one peer to another with the command *migrate* without stopping the runtime.

Listing 1.8: Consumer browser direct access

```
{
    "id": "consumer",
    "script": "icwe_browser.js",
    "browser": {
        "path" : "/map",
        "only" : true
    }
},
```

6 Conclusions

We presented Web Liquid Streams and how it can be used to develop mashups of streaming Web APIs. Like many data-flow based mashup tools, Web Liquid Streams uses pipelines to represent how information flows between Web data sources and the widgets visualising it. However, Web Liquid Streams data flow pipelines can have arbitrary topology and are used to continuously stream data so that the mashup widgets can be updated in real-time as more information is streamed through the mashup.

WLS runs both on Web servers and Web browser-enabled devices making it possible to implement the presentation layer on the Web browsers and dynamically spread the integration layers between the Web servers and the Web browsers. Moreover any peer attached to the application (i.e. sensors, browsers, ...) can become a Data producer, filter as well as consumer of a topology. During the 10 minutes of the rapid mashup challenge we demonstrated the main features of the framework: Static deployment of the mashup topology with JSON descriptions, iterative and incremental live development of a topology at runtime and the liquid [16] distribution of the mashup widgets on different Web browsers.

We are currently working on a visual topology editor tool, which would shift the building of a mashup from the textual editing of low-level JSON descriptions to a high level visual drag-and-drop. The tool would connect and interact with the already implemented REST API of the runtime for monitoring and deployment of the mashup topology.

Acknowledgment The work is supported by the Hasler Foundation with the Liquid Software Architecture (LiSA) project.

References

1. Zang, N., Rosson, M.B., Nasser, V.: Mashups: who? what? why? In: CHI'08 extended abstracts on Human factors in computing systems, ACM (2008) 3171–3176
2. Liu, Y., Liang, X., Xu, L., Staples, M., Zhu, L.: Composing enterprise mashup components and services using architecture integration patterns. Journal of Systems and Software **84**(9) (2011) 1436–1446
3. Aghaee, S., Nowak, M., Pautasso, C.: Reusable decision space for mashup tool design. In: Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems, ACM (2012) 211–220
4. Daniel, F., Matera, M.: Mashups: Concepts, Models and Architectures. Springer (2014)
5. Babazadeh, M., Gallidabino, A., Pautasso, C.: Decentralized stream processing over web-enabled devices. In: Proc. of 4th European Conference on Service-Oriented and Cloud Computing (ESOCC 2015), Taormina, Italy, Springer (September 2015)
6. Aghaee, S., Pautasso, C.: Mashup development with HTML5. In: 4th International Workshop on Web APIs and Services Mashups (Mashups 2010), Ayia Napa, Cyprus, ACM (December 2010) 10:1–10:8
7. Daniel, F., Matera, M., Yu, J., Benatallah, B., Saint-Paul, R., Casati, F.: Understanding ui integration: A survey of problems, technologies, and opportunities. Internet Computing, IEEE **11**(3) (2007) 59–66
8. Hirzel, M., et al.: A catalog of stream processing optimizations. ACM Comput. Surv. **46**(4) (March 2014) 46:1–46:34

9. Daniel, F., Koschmider, A., Nestler, T., Roy, M., Namoun, A.: Toward process mashups: Key ingredients and open research challenges. In: Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups. Mashups '09/'10, New York, NY, USA, ACM (2010) 9:1–9:8
10. Biörnstad, B., Pautasso, C.: Let it flow: Building mashups with data processing pipelines. In: Service-Oriented Computing-ICSOC 2007 Workshops, Springer (2009) 15–28
11. Oehme, P., Krug, M., Wiedemann, F., Gaedke, M.: The chrooma+ approach to enrich video content using HTML5. In: Proceedings of the 22nd international conference on World Wide Web companion. (2013) 479–480
12. Phuoc, D.L., Hauswirth, M.: Linked open data in sensor data mashups. In: Proc. of SSN09, CEUR (2009) 1–16
13. Babazadeh, M., Gallidabino, A., Pautasso, C.: Liquid stream processing across web browsers and web servers. In: Proc. of ICWE. Springer (2015) 24–33
14. Babazadeh, M., Pautasso, C.: A restful api for controlling dynamic streaming topologies. In: Proceedings of the companion publication of the 23rd international conference on World wide web companion, International World Wide Web Conferences Steering Committee (2014) 965–970
15. Aghaee, S., Pautasso, C.: Live mashup tools: challenges and opportunities. In: Live Programming (LIVE), 2013 1st International Workshop on, IEEE (2013) 1–4
16. Mikkonen, T., Systa, K., Pautasso, C.: Towards liquid web applications. In: Proc. ICWE2015, Rotterdam, NL, Springer

Appendix

Mashup Feature Checklist

Mashup Type	Logic mashups
Component Types	Logic components
Runtime Location	Both Client and Server
Integration Logic	Choreographed integration
Data Passing Logic	Direct data passing
Instantiation Lifecycle	Short-living

Mashup Tool Feature Checklist

Targeted End-User	Programmers
Automation Degree	Semi-automation or manual
Liveness Level	Level 4
Interaction Technique	Textual DSL and other (console)
Online User Community	None