A RESTful API
for Controlling Dynamic Streaming Topologies

Masiar Babazadeh
Faculty of Informatics, University of Lugano,
Switzerland
masiar.babazadeh@usi.ch

Cesare Pautasso
Faculty of Informatics, University of Lugano,
Switzerland
c.pautasso@ieee.org

ABSTRACT

Streaming applications have become more and more dynamic and heterogeneous thanks to new technologies which enable platforms like microcontrollers and Web browsers to be able to host part of a streaming topology. A dynamic heterogeneous streaming application should support load balancing and fault tolerance while being capable of adapting and rearranging topologies to user needs at runtime. In this paper we present a REST API to control dynamic heterogeneous streaming applications. By means of resources, their uniform interface and hypermedia we show how it is possible to monitor, change and adapt the deployment configuration of a streaming topology at runtime.

Categories and Subject Descriptors

D.2.11 [**Software**]: Software Architectures—*Patterns (pipeline)*

Keywords

RESTful API, Stream, Framework

1. INTRODUCTION AND MOTIVATION

Stream processing frameworks allow to build topologies to perform computations over infinite streams of data. These topologies can be distributed to run across multiple hosts and, with modern frameworks, can also dynamically change their structure as well as adapt the deployment configuration to make elastic use of the available resources. Most existing streaming frameworks make use of procedure calls [18] (remotely across the network or local inter-process calls) to manage the topology execution environment, monitor the behavior of the topology and dynamically modify/adapt its configuration.

In our work we are interested to study how to run stream processing topologies on the Web. We are building the Web Liquid Streams (WLS) framework for connecting multiple processing operators written in JavaScript into topologies that can be deployed both across multiple Web servers and Web browsers. In this paper we describe the architecture of Web Liquid Streams, particularly focusing on its RESTful design for managing and controlling the streaming topology.

The WLS framework takes advantage of modern capabilities of the HTML5 Web platform, which has enhanced the communication and interaction capabilities of Web browsers beyond the basic HTTP protocol. In particular, we use WebRTC [5] to stream data directly between Web browsers and WebSockets to stream data between browsers and servers. Whereas these low-level protocols clearly break the client/server and statelessness constraints of REST, our goal is to take advantage of the properties of REST to scale the size of the infrastructure on which the topology can be deployed, hence the need to introduce an API for the runtime system that follows the REST architectural style.

The main features of the RESTful API involve:

1. Resource identifiers to address the basic components of streaming topologies and their deployment and runtime configuration;

2. The Uniform interface applied to inspect, monitor and dynamically change the streaming topology;

3. Standardized representations: we use JavaScript for programming the processing operators and JSON for describing the topology.

4. Code on Demand: the streaming topology description references the scripts to be executed as part of its operators, which are then dynamically downloaded to be executed by the available execution resources.

5. Loose coupling between the main system components: the Peers, which consume each other's RESTful interface, and the Controllers which discover the state of each Peer through hypermedia.

6. Human-redable interface: Web browsers can navigate through the topology, inspect its execution state and even participate in the actual execution of some of its operators.

The contribution of this paper is twofold. First, it explores how to use the REST architectural style to design a management and control API for dynamic streaming topologies, which historically have always been configured using RPC. Second, it presents in detail the RESTful API design for the WLS framework for distributed stream processing over heterogeneous, Web-friendly devices.

In the rest of this paper we first introduce Web Liquid Streams in more detail (Section 2). Then we demonstrate how REST fits the needs of a dynamic heterogeneous streaming application (Section 3) and bring use cases (Section 4) and an example (Section 5). Finally we show related works (Section 6) and draw some conclusions (Section 7).

2. WEB LIQUID STREAMS

Web Liquid Streams (WLS) is a framework for building topologies of data streams that can run on both Web browsers and Web servers, which can be deployed on all kinds of hardware devices (from pervasive microcontrollers such as Arduino/Beaglebone, embedded PCs like Raspberry Pi or Tessel, as well as mobile smart-

phones and tablets). Given the heterogeneity of the targeted deployment environment, we chose JavaScript as the language for developing the processing operators and Node.JS [1] as the basic building block for the WLS runtime.

In streaming topologies, data flows from producers (e.g., sensors) to consumers (visualizations on mobile devices). In between, there can be all kinds of intermediate processing, filtering, data aggregation steps. Each stage of the streaming topology is executed by one or more independent worker, which can be elastically replicated as necessary. The topology, its deployment configuration, and the available execution resources (i.e., Peers) are all managed using the RESTful API described in this paper.

More in detail, behind the API, we use a hierarchical process structure, which is replicated on each host running WLS. The top-level element is the Peer process, which publishes the REST API, and controls the execution of its children processes through local IPCs. In REST terms, the Peer acts as a gateway, which maps the external HTTP protocol into a different protocol used to interact with local processes. Multiple Peers can be started on different hosts so that the processing capacity of the topology can grow dynamically.

The Peer can spawn one or more Operator, the building block of a topology. Operators are associated with a specific topology processing stage and with the corresponding script (in JavaScript) to be invoked for every stream element received. The actual execution of the script is performed by the Worker processes (replicas/instances of Operators), which are directly connected to upstream and downstream Workers using the appropriate protocol. To add parallelism within a particular Operator of the topology, multiple Workers for the same Operator can be dynamically started.

Each Peer also runs a Controller process that is in charge to check the integrity of the other Peers connected. If a Peer is faulty or not responsive, the Controller restarts the lost Operators and Workers on another available Peer by creating new Operators and redirecting the traffic there. If some Peers leave the topology, the Controller is in charge of reorganizing its deployment configuration accordingly. Moreover, the Controller is also in charge of dealing with faults and load balacing at Worker level: if an Operator becomes a bottleneck, the Controller will increase the parallelism by adding more Workers; if a Worker is faulty or dead, the Controller is in charge of restarting it.
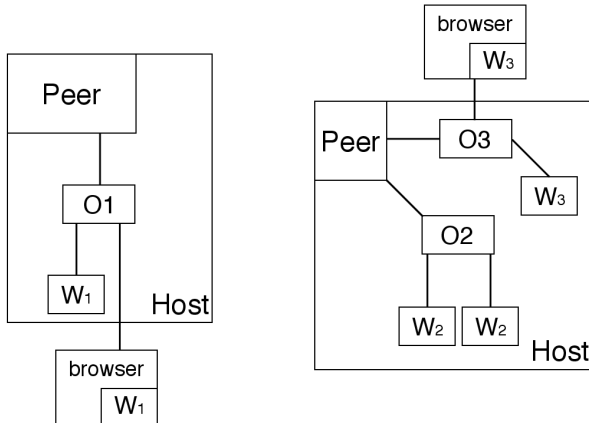

Figure 1: WLS process trees for two example deployments on a Web server host and on a Web browser

Figure 1 shows the processes running as part of a Peer. There are two Peers, on which a topology of three Operators has been deployed. Each Operator runs one or more Workers. Some Operators allow requests from Web Browsers and can outsource Workers on them. Security on Web browsers is achieved by sandboxing the execution environment.

Users can feed a topology referring to a set of scripts (written in JavaScript) to a Peer which will deploy them on the available resources (all known Peers). A topology is a description of the structure of the data stream that has to be run, in the form of a graph. The scripts instead describe how the stream elements should be processed at each node of the graph.
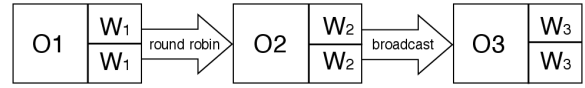

Figure 2: Logical view of the topology

Figure 2 shows the logical topology, abstracting its physical deployment shown above in Figure 1. The graph has Operators at the vertices, which spawn Workers to execute the scripts. Workers in O1 produce a stream of data which is forwarded to O2, whose Workers execute the script and send the result downstream to O3. The way data is shared among replicated workers is specified as part of the bindings of the topology. It can take the form of round robin or broadcast. With *round robin*, the data produced by an Operator is partitioned among the downstream workers running the following Node. This allows to scale-out processing of expensive Operators, since more workers can share its workload. With *broadcast*, a copy of the data produced by an Operator is sent to all downstream Workers associated with the following Operator. This is typically used with Operators that play the role of data consumer, for which multiple instances (e.g., on multiple browsers) can be dynamically started to observe the results of the streaming computation.

Overall, the approach of WLS is considered "liquid" [6] because as a fluid adapts its shape to the one of its container, so WLS will adapt the stream (liquid) to flow across the available hosts (pipes). As the pressure of the liquid increases (increased stream data rate), WLS will also adapt the shape of the pipe to avoid bottlenecks (elastic resource allocation to critical topology stages) so that the stream can fill each execution host according to its capacity.

3. RESTFUL API DESIGN

This section introduces the design for a REST API for controlling dynamic heterogeneous streaming topologies. The implementation of a REST API for WLS comes naturally as each Peer process is a Node.JS server which is able to serve HTTP requests. First we introduce the published resources, then we show how HTTP methods are used to access and modify their state and finally we show their representation media types.

3.1 Resources

The resources exposed by the API represent the most important concepts used to manage the deployment and the execution of a streaming topology in WLS. In the following we introduce the URI template and the informal semantics and hypermedia relationships between the resources. The hypermedia graph of the API is also summarized in Figure 3. The Figure shows the hyperlink connections between representations returned by performing `GET` requests on the corresponding resource.

/
The Peer root resource provides hyperlinks to the other top-level resources so that hypermedia can be used to dynamically discover what feature each of the WLS peer can offer.

/peers
This resource represents the collection of Peers known by a Peer and allows Peers to discover and refer to each other.

/peers/:pid
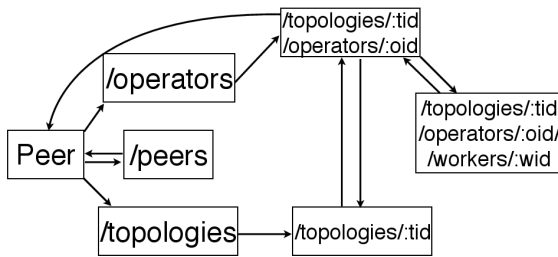This resource represents the information about a Peer that is known

Figure 3: Hypermedia navigation map, showing the resources that can be discovered from each GET request.

by the Peer from which it is retrieved. It also includes the hyperlink to the root of the REST API of the peer.

/operators
The Operator resource collection represents the set of all Operators (from all topologies) deployed on the Peer.

/topologies
The topology resource collection represents the set of topologies known by the Peer.

/topologies/:tid
A specific topology represents how Operators are interconnected to form a data stream. It is used to deploy a new topology into the system as well as to dynamically modify it and control its state of execution. A topology representation contains hyperlinks to its Operators, which are addressed as sub-resources for namespacing purposes.

/topologies/:tid/operators
The Operators collection of a topology.

/topologies/:tid/operators/:oid
Each Operator resource represents the individual processing step within a topology and contains the actual script to be executed. It also contains hyperlinks to connected Operators that allow to follow the topology.

/topologies/:tid/operators/:oid/workers
Each Operator resource allows to manage its worker collection (start, stop, migrate workers).

/topologies/:tid/operators/:oid/workers/:wid
The worker resource represents the execution state of an individual worker thread and allows the Controller to retrieve monitoring information about its performance.

3.2 Uniform Interface

For each of the previously introduced resources, we specify the semantics of applying one of the HTTP methods to it. If a method is not mentioned, a default `405 Method not allowed` response can be expected.

3.2.1 Root
GET /
Retrieves a list of hyperlinks to the contents of the Peer. There are three main links: /peers, /topologies, /operators. Additionally, some summary statistics about the peer performance are included.

3.2.2 Resource Management and Peer Discovery
GET /peers
Retrieves the collection of Peers known to the Peer. The collection includes both relative hyperlinks to the local Peer resource identifiers (/peers/:pid) as well as to the absolute hyperlinks to the REST API of the known Peers (http://ip:port/). Finding a Peer listed in the collection does not mean that the Peer has established

an actual connection to it for the purpose of streaming data, but only that the Peer is known. If a connection has been established, the list contains the last CPU usage value seen on that Peer.

POST /peers
A `POST` request on the /peers path with a payload referencing the address (IP:port) of the Peer informs the receiver that a new Peer exists on the network. The receiver stores the Peer data in the collection and returns the updated list of known Peers.

GET /peers/:pid
Retrieves the state of the Peer and a hyperlink to its REST API.

DELETE /peers/:pid
Used to remove a Peer with id `pid` from the list of known Peers.

DELETE /peers/:IP:port
Used to remove the Peer with address `IP:port` from the list of known Peers.

3.2.3 Topology Management
GET /topologies
Retrieves the list of topologies started from this Peer with hyperlinks to their resource identifier in the form of /topologies/:tid.

GET /topologies/:tid
Retrieves the current execution state of the topology with id `tid`. The result shows the topology specifying which Operators are running where and which script are they running. Hyperlinks to each Operator are also included.

DELETE /topologies/:tid
This method shuts down a topology.

POST /topologies
This method is used to create a new topology. The payload represents the structure of the topology to be implemented.

PUT /topologies/:tid
This method is used to create a new topology and associate it with the given identifier. The payload represents the the structure of the topology to be implemented.

3.2.4 Operator Configuration
GET /operators
Retrieves the list of all Operators deployed on this Peer with hyperlinks to their resource identifier.

GET /topologies/:tid/operators
Retrieves the list of Operators running on this Peer for topology with id `tid`.

GET /topologies/:tid/operators/:oid
Retrieves the representation of the Operator with id `:oid`. The representation includes the list of Workers running (with hyperlinks to contact them), the script that they are running, the connections they have, as well as a hyperlink back to the topology it is part of and information about the overall performance (for example, the request/response rate or the CPU usage aggregated across all of its workers).

PUT /topologies/:tid/operators/:oid
Performing the request with a payload carrying a script and the bindings creates an Operator named `oid` for the topology `tid`. This operation is not supported by `POST` as the name of the Operator has to be know a priori in order to perform the bindings described in the topology. Workers created in this Operator will run the script and performs the connections specified in the bindings. If the Operator identifier already exists, it is updated with the new information. This requires to stop the workers, update the script and the stream connections and then start the workers with the new script.

PATCH /topologies/:tid/operators/:oid/script
This request with a payload linking a new script updates at runtime

the current script Workers are running with a new version of it, without modifying the connections they have.

PATCH /topologies/:tid/operators/:oid/bindings
This request with a payload referencing new connections updates the bindings. In this case the overall topology is modified at runtime.

DELETE /topologies/:tid/operators/:oid
Stops the Operator with id `oid`.

3.2.5 Worker Configuration

POST /topologies/:tid/operators/:oid/workers
The request creates a new worker, the payload is not necessary as the Operator already has all the information for its creation (that is, script to be run and connections to make).

GET /topologies/:tid/operators/:oid/workers
Retrieves the list of Workers running on the Operator with id `oid`. The list contains hyperlinks to contact every Worker.

GET /topologies/:tid/operators/:oid/workers/:wid
Retrieves the status of the Worker with id `wid`. The result includes uptime, and information about the Worker performance (for example, request/response ratio, throughput).

DELETE /topologies/:tid/operators/:oid/workers/:wid
Deletes the Worker with id `wid` from the Operator by stopping it and removing its connections and deleting it.

POST /topologies/:tid/operators/:oid/browsers
Used to create a Worker for Operator with id `oid` on a browser. Returns a Web page and a script to be run. It only works if browser flag is specified in the topology description.

3.3 Representations

3.3.1 Operators

A collection of Operators is either returned when performing a `GET` request on `/operators` or on `/topologies/example/operators`.

```
{
  "operators" : [
      {
        "topology" : "example",
        "id" : "a",
        "workers" : [...],
        "CPU usage" : "50%",
        "href" : "/topologies/example/
            operators/a",
        "peer" : "http://IP:port/",
        "replicas" : [
  "http://IP2:port2/topologies/example/
    operators/a"
        ]
      }
  ]
}
```

Listing 1: Example collection of Operators

Listing 1 shows an example collection of one Operator. The JSON format is custom, as it is less verbose. Through content negotiation it is possible to retreive different kind of representations. The array contains objects representing the Operators, defining the id of the Operator, the hyperlink to contact it and the name of the topology it is part of. A link back to the Peer on which the Operator is deployed is also provided. This allows to conveniently aggregate the Operator configuration of multiple Peers. Moreover, if the Operator had to be replicated on other Peers, due to overloading, an array with direct hyperlinks to the replicas is provided.

3.3.2 Stream Topology

Listing 2 shows the topology schema that describes the JSON to be sent to the Peer when creating a new topology through a `PUT /topologies/:tid` request. The Operator ids may be given by the user and are used to create the corresponding resource identifiers.

The `operators` key contains an array of objects describing which Operator runs which script and the number of Workers that should be started in the initial deployment configuration of the Operator. If no number of Workers is provided, the Operator will be started with one Worker by default. The `browser` key specifies whether browsers can connect to this Operator and become part of the topology, while the `max-workers` and `min-workers` keys define the maximum or minimum amount of Workers that can be run for that Operator. By default, the minimum is one and the maximum depends on the available execution resources.

The value of the `bindings` key instead contains an array of objects describing the bindings, that is, the flow of the data stream between Operators. The objects also specify the algorithm used to send the data to multiple workers. WLS currently offers Round Robin and Broadcast as possible routing algorithms.

```
{
  "title": "Topology Schema",
  "type": "object",
  "properties": {
    "id": {
      "type": "string"
    },
    "operators": {
      "type": "array",
      "minItems" : 1,
      "items" : {
        "type" : "object",
        "properties" : {
          "id" : { "type" : "string" },
        "workers" : { "type" : "integer" },
        "browser" : { "type" : "boolean" },
    "max-workers" : { "type" : "integer" },
    "min-workers" : { "type" : "integer" }
      },
    }
    },
    "bindings": {
      "type": "array",
      "minItems" : 0,
      "items" : {
        "type" : "object",
        "properties" : {
        "from" : { "type" : "string" },
          "to" : { "type" : "string" },
        "type" : { "type" : "string" },
      }
    }
  }
}
```

Listing 2: Stream topology schema

3.3.3 Operator Configuration

When executing a `PATCH` request to patch an Operator, a JSON payload is sent containing the description of the change to apply. There are two kinds of patches: to modify which script is associated with the Operator on the `.../script` sub-resource or a modification of the bindings (that is, the topology) at runtime on `.../bindings`.

```
{
  "bindings" : {
    "from" : "/topologies/example/operators/
        c",
    "to" : "/topologies/example/operators/a"
  }
}
```

Listing 3: Updating the bindings of Operator b at runtime

Listing 3 shows the JSON sent to update the bindings of Operator b, reversing the flow of the topology. The object contains a `from` key and a `to` key whose values are different from the previous binding of the Workers. It is mandatory to define at least one of the two in order to update a binding (the one not defined keeps the old binding). If the Operator is supposed to become the final stage of the topology, then it is sufficient to leave empty the string in the `to` key. Likewise, viceversa for Operators that should be moved at the beginning of the topology (empty `from`). A similar JSON needs to be sent to the other Operators a and c to update their connections as well.

3.3.4   Worker state monitoring

Listing 4 shows the result of a `GET` request on the `/topologies/example/operators/a/workers/0` path. It contains the id of the Worker, information about how to contact it again and a hyperlink to its Operator. Performance information includes how long has it been up, the total number of messages that have been processed since it was started, as well as its request/response ratio for the last second. The latter is very important for the Controller in order to detect whether the Operator is a bottleneck in the topology (incoming request messages / outgoing response messages > 1).

```
{
  "worker" :
    {
      "id" : "0",
      "href" : "/topologies/example/
          operators/a/workers/0",
      "operator" : "/topologies/example/
          operators/a",
      "uptime" : "3600",
      "messages" : 42,
      "req-res-ratio" : 1.5
    }
}
```

Listing 4: Worker state returned as JSON

4.   USE CASES

4.1   New Peer Joins the Network

When a new Peer P joins the network, it will contact another Peer through a `POST` request on `/peers`. The payload of the request references the address of P. The Peer receiving the request updates its known Peer list and replies with it to P. The list not only features the address of known Peers, but also their CPU usage if known. P then chooses which Peer to contact giving preference to the most available ones (least CPU usage). This Peers subset is then contacted through the same interface; the process stops after enough Peers are contacted. The idea behind this procedure is to gather a sufficient amount of Peers to run the topologies that will be deployed on P. Contacting busy Peers (i.e., CPU usage greater than a fixed threshold) is not worth as they will not have capacity to run Operators.

4.2   Setting up a Topology

To setup a topology users have to perform a `PUT` request with a payload that contains the description of the topology which includes the links to download the scripts to run for each Operator. The Peer receiving the topology description fetches the scripts spreads them on the least used Peers it knows by checking their availability. The Peers will receive the setup though a `PUT` request on `/topologies/example/operators/a` with a payload referencing the script and the connections to be performed. This automatically starts one Worker that is ready to process the stream. Users can also specify, in the description, the number of Workers to be initially started at a specific stage of the topology.

4.3   Browser as a Worker for an Operator

When a Web browser wants to join a topology, it contacts a Peer through its API it can navigate to discover an Operator of interest using `GET` requests. Once it reaches the Operator `.../operators/c`, a specific representation is retrieved to display some information about the purpose of the Operator together with an HTML form. If the user would like to start processing the Operator using the browser, the form is submitted through a `POST` request on `.../operators/c/browsers`. P registers the browser as a Worker for Operator c running for the topology example. The response contains the script to be run using a WebWorker thread in the browser. Communication of the data stream with the rest of the topology is proxied by the Peer through a Websocket channel, or it is established using a WebRTC data channel if the upstream or downstream workers are also running in a Web browser. If the browser is disconnected (e.g., the tab running the worker is closed by the user) the channel is closed and the worker is stopped. If the `browser` flag is not specified in the Operator configuration, the result of the `POST` will be a 405.

4.4   Perform New Binding

In the case in which an Operator N needs a new binding (i.e. a new Operator has been added after N in the topology), the Peer sends a `PATCH` request on `.../operators/N/bindings` to the Operator with a payload specifying the new connections. The Operator automatically redirects the traffic of its Workers to the new Operator with the connections received through the request, thus changing the topology at runtime.

4.5   Load Balancing

Load balancing is performed by the Controller, which calculates the efficiency of the Operator by accessing the state of the Workers. The Controller has to perform a `GET` request on the list of Workers `.../operators/slow_operator/workers`, and with the data received it is able to compute the efficiency of the Operator and detect if there is a bottleneck. If that is the case, the Controller is going to add more Workers by performing a `POST` request on `.../operators/slow_operator/workers`. In the case in which there is some idle Worker, the Controller may decide there are too many of them, thus it shuts an appropriate number down performing a `DELETE` request on the corresponding worker resource. In the case of slow Web browsers, the Controller can try to add Workers on the browser. If the Workers keep being slow, the controller cannot add more Workers on it.

4.6   Fault Tolerance

We consider three different fault types: a Peer failed, an Operator failed or a Worker failed.

Peer Failed: Peers that started a topology should be aware of the status of other Peers that host part of their topologies. This is done by

the Controller which polls their status through a `GET` request on the root path. In the case of slow response or no response at all, the Controller may decide to no longer rely on the unresponsive Peer by moving the computation on another available Peer. It does so by first issuing a `PUT` request on `topologies/example/operators/a` on the available Peer to recreate the Operator, with a payload referencing the script and the bindings, as well as the last number of Workers observed. This way the configuration of the Operator in the failed Peer is reproduced in the new Peer. This has to be repeated for all operators that have been deployed on the failed Peer. When the failed Peer becomes reachable once again, the Controller will send a `DELETE` request on `/topologies/examples/operators/a` to shut the old Operator down.

Operator Failed: The Controller also checks the availability of Operators running on topologies set up by their Peers. The Controller is interested in the status of the Operators because they may have crashed even if the Peer is still responding. By inquiring about their state through a `GET` request on `/topologies/example/operators/a`, it receives the current status of the interested Operator. If the Peer replies that the Operator is unavailable, the Controller issues a `DELETE` request on the same Peer using the URI of the failed Operator. This way, the Operator is removed from the topology. Then, the Controller will attempt to redeploy and restart the Operator with a `PUT` request with a payload referencing the script and the connections peformed by the dead Operator as well as the same number of Workers. In this way a new Operator is created to replace the broken one.

Worker Failed: The Controller should also be in charge of checking for failed Workers. If a Worker does not answer the `GET` request on `topologies/example/operators/a/workers/wid0`, the Controller assumes it failed. If that is the case it will start a new Worker on the Operator with `POST` and send a `DELETE` request to remove the unresponsive Worker.

4.7 Operator Overloaded

If a Controller wants to add more Workers, but the Peer is reaching its maximum capacity, the Controller will choose another known Peer with available capacity and use a `PUT` request on `/topologies/example/operators/a` to deploy a replica of the Operator. This process offloads the rest of the computation for that Operator on another Peer. If more Workers are needed, they will be created on the offloaded Operator replica. The full Operator and the replicated one will have in their representation the `replica` key pointing to each other, thus the replicas can be easily recognized.

4.8 Migration

The process of migration happens when the physical host of a Peer P (be it a cluster of machines, a pervasive device or a server) has to leave the topology. In this case P executes a `DELETE` request on all the Peers it knows on the `/peers.IP:port` path, specifying its own IP:port in the path. This causes the Peers receiving the request to check if P is part of their topologies. If that is the case, then the Controllers will search for an available Peer to substitute P, and will recreate Operators and Workers there through `PUT` and `POST` requests. Once this is done, they will proceed to remove the Operators on P and remove P from their list of known Peers. When no more Operators are running on P, it can safely leave the network.

5. EXAMPLE APPLICATION

Imagine a streaming application that gathers the video streams from one or more webcams or mobile phone cameras, execute face recognitions, applies a filter and combines them into a single
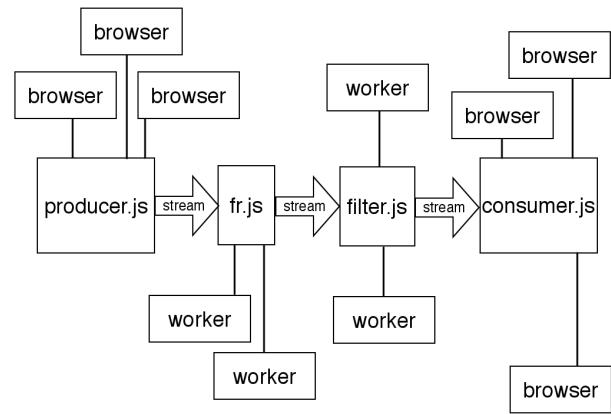

Figure 4: The example webcam mashup topology.

view. This can be implemented with WLS using a topology of four Operators: one dedicated to gathering video feeds, one for face recognition, one that applies a filter to the snapshots and one which aggregates the filtered videos into a single frame.

Listing 5 shows a setup JSON file which is sent as a `PUT` request to the `/topology/webcam-mashup` resource. Browsers can run the first and last Operators of the topology.

Figure 4 shows the topology described in the JSON topology provided, with Operators identified with the scripts they run. Browsers are connected to the Operator running `producer.js` and the one running `consumer.js` while server-side Workers are executing the intermediate `fr.js` and `filter.js` Operators. As the face recognition script is computationally heavy, it may require a dedicated server to run. After Peer P received `PUT` request with a payload referencing the discussed JSON, it has to find available Peers to run the Operators and perform `PUT` requests to star them. When a browser contacts the Operator running `producer.js` through `.../producer/browsers`, it receives back a script that will connect the browser to the topology so that the data can flow through it. In this case, the webcam video will be captured and sent frame by frame through the data stream to the next Operator. The `fr.js` script takes the incoming video frames and executes a face recognition procedure on the frames. The result of the computation is the name of the persons in the frame as well as their position; both are forwarded, with the frame, to the following Operator, which runs `filter.js`. The script takes the incoming video frames and applies some image filter (e.g., negative, B/W, HDR, tilt shift) and sends the result to the final stage of the topology. In this stage, the `consumer.js` aggregates the images received and displays them on the Web browser with the names of the recognized faces.

We can imagine the work of `fr.js` becoming heavier as more producers join the topology. If that is the case, the Controller by performing a `GET` request on `.../face_recognition/workers`, notices that the request/response rate of the Worker is greater than 1, which means the Operator itself is a bottleneck of the topology. To fix the problem, the Controller adds some Workers by performing `POST`s requests on `.../face_recognition/workers`. Workers will be created, increasing the parallelism of the `fr.js` stage of the topology, solving the bottleneck issue.

To check the overall status of the topology, the Controller in P polls data from the Peers and Operators involved in the topology. While the `fr.js` Operator is increasing the number of Workers, it starts exhibiting a faulty behaviour. To cope with that, the Controller can restart the Operator by sending a `PUT` request to `/topologies/webcam-mashup/operators/face_recognition`

to the same Peer with a payload referencing the script `filter.js` and the same connections. Unfortunately, the Controller may notice that the Peer starts exhibiting a fault behaviour as well. In this case, the Controller may decide to offload the Operator to another Peer by sending the same request to it to create the Operator and the Workers. Once this is done and Workers are connected correctly, the Controller sends a `DELETE` request to the previous Operator to remove it.

After the new Operator has been set up, the dataflow keeps increasing as more users join the topology. The critical stage is `fr.js` as it has to perform the most heavy computations. The Controller can add Workers up to the point in which the Peer is full and can not handle more, but still the Operator may be slow. To solve this problem, the Controller will choose an available Peer among the list of known Peers and perform a `PUT` request on `.../operators/face_recognition` with a payload referencing `fr.js` and the connections the crowded Operator has. This creates a new Operator on a different machine which performs the same computation as the crowded one, easing the computation from it.

```
{
  "topology" : {
    "id" : "webcam-mashup",

    "operators" : [
      {
        "id" : "producer",
        "script" : "producer.js",
        "browser" : true
      },
      {
        "id" : "face_recognition",
        "script" : "fr.js",
      },
      {
        "id" : "filter",
        "script" : "filter.js"
      },
      {
        "id" : "consumer",
        "script" : "consumer.js",
        "browser" : true
      }
    ],

    "bindings" : [
      {
        "from" : "producer",
        "to" : "face_recognition",
        "type" : "roundrobin"
      },
      {
        "from" : "face_recognition",
        "to" : "filter",
        "type" : "roundrobin"
      },
      {
        "from" : "filter",
        "to" : "consumer",
        "type" : "broadcast"
      }
    ],
  }
}
```

Listing 5: Topology for a webcam mashup application

Figure 5 shows a crowded Operator. The Operator itself has many Workers, and this implies busy CPU for the Peer. The depicted `PUT` request offloads part of the computation on another available Peer.

When the entire topology has to be shut down, a single `DELETE` request on `/topologies/webcam-mashup` on Peer P is sufficient to stop it. This operation triggers subsequent `DELETE` requests going to the Workers first, and then to the Operators shutting them down. Finally, when all the Operators are removed, the topology itself is removed from P.

6. STATE OF THE ART

In the last decade, many streaming frameworks have been proposed. Older systems were static and homogeneous, e.g., Aurora (2002) [7, 8, 20], StreamIt (2003) [17], CQL (2003) [3] and Sawzall (2003) [14], These systems did not implement any sort of control over the structure of the running topology. More recent systems, like DryadLINQ (2008) [19], Storm (2011) [2] and TimeStream (2013) [15] instead introduced control over the topology at runtime [4]. The implementation of control structures for these systems has been proposed with RPC.

RPC is well suited to execute calls on remote machines, thus to update something on an Operator, a procedure can be executed on the Operator in order to update it. The programming model of RPC makes remote calls appear as local calls. Thus, it hides latency issues as well as partial failures. For example, if a failure happens, clients do not know if it happened before their message arrived, or while the message was processing, or if the error is the result of the computation. Moreover, in RPC-oriented systems, a new service interface add a new interface protocol, that is the consumers have to hard-code knowledge of method names and semantics and must inherently know which method to call and in which order while there is no semantic constraints on methods. REST on the other hand solves many of these problems [18]. HTTP status codes for example help in assigning blame. The uniform interface constraint enables visibility into interactions. Moreover, both parts can evolve independently keeping the interface the same. The representation format is not method specific as with RPC-oriented approaches, thus through Media Types we can give standardized representation to resources.

Another software connector relevant to this paper is the data stream. There has been some work to extend REST with support for data streams to deal with different problems, mostly centered around the notification of resource state changes. In the following introduce some of these approaches in relationship with our work.
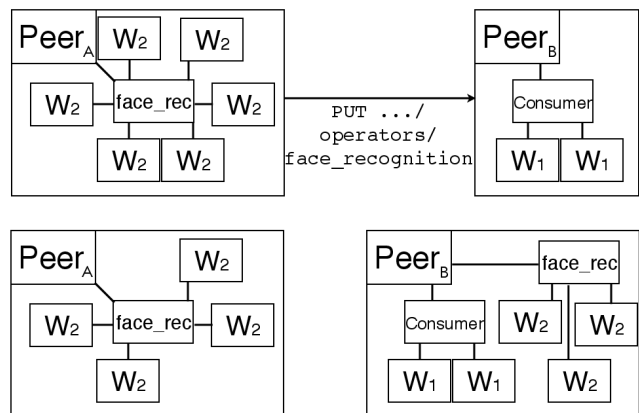

Figure 5: A crowded Peer A contacts another Peer B (top) to offload his busy Operator and part of the workers there (bottom).

6.1 REST vs. Observer Pattern

In [16] the authors show how to support real-time notifications with resource oriented architectures using REST observers. The main idea is to allow Web clients to receive events about state changes in real time thanks to an observer that monitors the server and keeps track of changes. The concept can be applied to our design, especially to handle the relationship between Workers and Controllers. This way Controllers would not need to POLL Workers to access their states and take decisions.

6.2 Server-to-server change propagation

Regarding server-to-server change propagation, when it comes to contact the Controller from the Workers or to exchange status updates between known Peers, one example is PubSubHubbub [11]. It is a simple server-to-server publish/subscribe protocol based on webhooks for any web accessible resource. When a server publishes (or updates) a resource, it also posts an update to a hub which stores the new changes. When a resource is fetched from the server, it specifies in the response Headers a link to the hub, thus who fetched the data can avoid the process of continuous polling by subscribing to the hub for the interested data. Each time the server publish a new update, the hub will notify all the subscribers.

Functional Observer REST (FOREST) [9] is another mechanism that allows Web resources (or objects) that are interested in the state of other resources to be notified when that state changes, and are able to update their own state accordingly. The interaction pattern of the protocol propagating the changes is very similar to PUSH: the state is set as a Function of its current state (plus other object states that it observes) and the observation occurs through either PULL or PUSH of the linked object state.

6.3 Server-to-client change propagation

Server-to-client change propagation can be useful when, in our case, the Operator has to contact a Worker running on a Web browser. One broadly adopted solution is Asynchronous JavaScript and XML (AJAX) [13], which relies on the so-called long-polling technique. The client opens an AJAX connection with the server, which keeps it open so that updates on the interested resource are sent as soon as they happen. Another approach for a server to push data to a browser are Server-Sent Events [12] which are typically streamed to the recipients. To be able to receive these events, the client must know the resource which is emitting the events, or has to become aware of that resource via a script. WebSocket [10] is another technology available to enable interactive communication between a browser and a Web server using HTTP only during handshake. It enables bi-directional communication and connection management, clients can send messages to the server and receive response without polling it. This is the approach currently used in WLS to stream data from/to workers running on a Web browser.

7. CONCLUSIONS

In this paper we presented a RESTful API for controlling dynamic heterogeneous streaming systems. The API we propose has been originally defined for Web Liquid Streams, a novel programming platform for dynamic heterogeneous streaming systems. We believe that with the adequate adjustments, the proposed API can fit any similar system. The main idea is to provide a REST interface that supports the creation, monitoring and adaptation of stream processing topologies that can be deployed across a heterogeneous set of execution hosts (including both server-side Web servers and client-side Web browsers). The resulting RESTful API provides a set of basic resources for peer management and addressing, topology definition and dynamic modification, as well as Operator execution

monitoring, allocation, and migration. The API is used to build a monitoring and control systems to ensure the automatic load balancing and fault tolerance of the streams.

8. REFERENCES

[1] Node.js framework, 2009. `http://nodejs.org/`.

[2] Storm, distributed and fault-tolerant realtime computation, 2011. `http://storm-project.net/`.

[3] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[4] M. Babazadeh and C. Pautasso. The Stream Software Connector Design Space: Frameworks and Languages for Distributed Stream Processing. In *Proc. of WICSA*, 2014.

[5] A. Bergkvist et al. Webrtc 1.0: Real-time communication between browsers. *Working draft, W3C*, 2012.

[6] D. Bonetta and C. Pautasso. Towards liquid service oriented architectures. In *Proc. of WWW*, pages 337–342, 2011.

[7] D. Carney et al. Monitoring streams: a new class of data management applications. In *Proc. of VLDB*, 2002.

[8] M. Cherniack et al. Scalable Distributed Stream Processing. In *Proc. of CIDR*, 2003.

[9] D. Cragg. Forest: An interaction object web. In C. Pautasso and E. Wilde, editors, *REST: From Research to Practice*. Springer, 2011.

[10] I. Fette and A. Melnikov. The Websocket protocol. 2011.

[11] B. Fitzpatrick and B. Slatkin. PubSubHubbub Core 0.3. Technical report, Feb. 2010.

[12] I. Hickson. Server-sent events. *Candidate Recommendation CR-eventsource-20121211*, 2012.

[13] A. Mesbah and A. van Deursen. An Architectural Style for Ajax. In *Proc. of WICSA '07*, pages 9–.

[14] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*, 13(4):277–298, 2005.

[15] Z. Qian et al. TimeStream: reliable stream computation in the cloud. In *Proc. of EuroSys*, 2013.

[16] V. Stribu and T. Aaltonen. Enabling real-time resource oriented architectures with rest observers. In C. Pautasso, E. Wilde, and R. Alarcon, editors, *REST: Advanced Research Topics and Practical Applications*. Springer, 2014.

[17] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proc. of CC*, 2002.

[18] S. Vinoski. RPC and REST: Dilemma, disruption, and displacement. 12(5):92–95, Sept. 2008.

[19] Y. Yu et al. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. of OSDI*, 2008.

[20] S. Zdonik et al. The Aurora and Medusa Projects. *IEEE Data Engineering Bulletin*, 26:3–10, 2003.