Comparing a polling and push-based approach for live Open Data interfaces

Brecht Van de Vyvere[0000-0002-7671-6203], Pieter Colpaert[0000-0001-6917-2167] and Ruben Verborgh[0000-0002-8596-222X]

IDLab, Department of Electronics and Information Systems, Ghent University – imec
firstname.lastname@ugent.be

Abstract. There are two mechanisms for publishing live changing resources on the Web: a client can pull the latest state of a resource or the server pushes updates to the client. In the state of the art, it is clear that pushing delivers a lower latency compared to pulling, however, this has not been tested for an Open Data usage scenario where 15k clients are not an exception. Also, there are no general guidelines when to use a polling or push-based approach for publishing live changing resources on the Web. We performed (i) a field report of live Open datasets on the European and U.S. Open Data portal and (ii) a benchmark between HTTP polling and Server-Sent Events (SSE) under a load of 25k clients. In this article, we compare the scalability and latency of updates on the client between polling and pushing. For the scenario where users want to receive an update as fast as possible, we found that SSE excels above polling in three aspects: lower CPU usage on the server, lower latency on the client and more than double the number of clients that can be served. However, considering that users can perceive a certain maximum latency on the client (MAL) of an update acceptable, we describe in this article at which MAL point a polling interface can be able to serve a higher number of clients than pushing. Open Data publishers can use these insights to determine which mechanism is the most cost-effective for the usage scenario they foresee of their live updating resources on the Web.

Keywords: Web API engineering, Performance and Scalability, Open Data.

1 Introduction

The Open Data deployment scheme [1] defines 5 steps that data publishers can undertake to raise the technical and semantical interoperability of their Open datasets on the Web. With the use of the Hypertext Transfer Protocol (HTTP) as a communication protocol, a dataset becomes technically interoperable with the Web of data [2,3]. This allows Open Data consumers to retrieve a resource (e.g. a document that is part of the dataset) by sending an HTTP GET method to the Uniform Resource Identifier (URI) of the resource. For live changing resources, such as the measurements of a sensor, there are two communication mechanisms to share an update in a timely fashion to clients. First, there is pull where the client initiates the action to retrieve a resource. This category has two representatives: HTTP polling and HTTP long polling. Next, you have

push where the server pushes updates of a resource to the client. Server-Sent Events (SSE) and Websockets are implementations for this mechanism. Lubbers et al. [4] compared Websockets with HTTP polling for a dataset that updates every second and up to 100k clients, but this was only a theoretical analysis of the bandwidth usage and latency. Depending on the size of the header information, a bandwidth reduction of 500:1 can be made with Websockets and a latency reduction of 3:1. Pimentel et al. [5] went a step further and investigated how the physical distance between publisher and consumer impacts the overall latency. They performed a comparison between polling, long polling and Websockets where a new sensor update of roughly 100 bytes is published per second. They defined formulas to check when polling or long polling is feasible for updates on time, dependent on the network latency. When the network latency exceeds half the update rate of the dataset, then Websockets is the better choice. However, there is no evaluation performed on the performance of the server under a high load of clients, which is an important factor that needs to be considered for Open Data publishing.

One of the key features of publishing data on the Web is HTTP caching, which has not been addressed in related work [4,5,6]. This allows a resource to become stateless and can be shared with proxy caches or Content Network Delivery (CDN) services to offload the server. With push-based interfaces like Websockets, caching of a resource is not possible as the server needs to actively push the content in a stateful manner to all subscribed clients. In previous work [7], a minimum set of technical requirements and a benchmark between pull (HTTP polling) and push (Websockets) have been introduced for publishing live changing resources on the Web. This benchmark tested with only 200 clients, which did not yield conclusive results on the latency or scalability issues that arise inside an Open Data ecosystem, where client numbers of 15 000 are not an exception [8]. In this article, we will run a similar benchmark between HTTP polling and SSE with up to 25k clients. SSE is tested instead of Websockets, because it has a similar performance [6], communication is unidirectional which is suitable for Open Data and lastly, it only relies on HTTP instead of a Websockets protocol, which lowers the complexity of reusing a dataset.

The remainder of this article is structured as follows: we will provide in the related work section an overview of publication techniques and the current state of publishing RDF Streams on the Web. RDF Streams are applicable for describing live updating resources with the Resource Description Framework (RDF) and thus resolve the fourth step of the Open Data deployment scheme [1]. Thereafter, we conduct a field report to quantify how many live Open datasets are available on the Open Data portals of Europe and the U.S. to observe which update retrieval mechanism is used in different domains. In the problem statement section, we define our research questions and hypotheses, which we will then evaluate with a benchmark between HTTP polling and SSE. In the discussion and conclusion, based on the results of the benchmark, we will propose some guidelines for data publishers when to use pull or push interfaces.

2 Related work

2.1 Web publication protocols

HTTP polling. A client sends an HTTP GET request to the server, waits for a certain time interval after retrieving the response and starts again with requesting the resource. The benefit of this approach is that a resource becomes stateless and HTTP caching is possible. However, there is no strict guideline on how clients should time their request. For variantly updating resources, a client can not predict when the next update will be available. A higher polling frequency can minimize the latency on the client, but this comes at a higher bandwidth cost [4].

HTTP long polling. With long polling, the server only returns a response when a new update is available. This way, a client does not send redundant requests like HTTP polling. Also, the client does not wait before sending a request again. A resource becomes stateful as the server needs to maintain all the connections open. Pimental et al. [5] showed that long polling can have a similar performance as Websockets when the underlying network latency is lower than half the data measurement rate.

Server-Sent Events. With the growing demand for (near) realtime applications, HTTP is extended in 2014 with the support of Server-Sent Events (SSE). Similarly to long polling, the server holds the connection open for every client, but this remains open for pushing multiple updates instead of one. With the use of the EventSource API (supported by all browsers except IE and Edge), clients can receive updates of a resource in an event-driven fashion. Using SSE over HTTP/1.1 has the disadvantage that every requested resource requires a separate TCP connection, which can run into the limited number of connections a browser can open per domain. However, this is solved for servers that support HTTP/2, which multiplex all requests and responses over one connection.

Websockets. The Websocket protocol provides a bidirectional communication channel over one TCP connection for every client. HTTP is used to set up a handshake between client and server for transmitting data, but further communication happens over a raw TCP connection. The WHATWG Websocket standard describes how messages can be pushed between client and server, but there are also sub protocols (MQTT [9], CoAP, etc.) for more advanced features, for example a publish/subscribe broker to receive or send updates of a specific resource. Websockets has a similar performance as SSE [6], but a lower transmission latency when the server needs to send large messages above 7.5 kilobytes. Also, for client to server communication provides Websockets a lower transmission latency [6] than using HTTP.

WebSub. The WebSub [10] specification extends the communication pattern between clients and servers from above protocols with a third actor hub. A resource can be retrieved from the publisher (server), but consumers can also subscribe for updates through a hub instead of polling the resource URL. A resource is coupled with one

topic, which is exposed by one or more hubs for fault tolerance. Reusers of the data can receive updates by setting up a Web accessible server and subscribing to a topic. Hubs then send updates through HTTP POST requests (Webhook mechanism) to this server. While Open Data publishers can benefit from distributed hubs in an Open Data ecosystem, for example a hub can be reused by multiple data publishers, Open Data reusers are required to deploy a Web server to receive updates they are interested in. To enable the use case of autonomous intelligent agents [11] that wish to retrieve updates of a resource through a topic, then a service for subscribing to a topic must be made available. As this service will also need to decide on exposing polling or pushing to agents, we will not further elaborate on WebSub in this article.

2.2 RDF Streams

Arasu et al. [12] defines a stream S as a (possibly infinite) bag (multiset) of elements $\langle s, \tau \rangle$ where s is a tuple (the actual data without the timestamp of the element) belonging to the schema of S and $\tau \in T$ is the timestamp of the element. The Resource Description Framework (RDF) Stream Processing (RSP) Community Group, which focuses on processing RDF-modelled data, has applied this definition for RDF streams [13] where an RDF stream S is a (potentially) unbounded sequence of timestamped RDF statements in non-decreasing time order. TripleWave [14] is a tool that transforms Web streams, which only differs from an RDF stream by its data model [7], into RDF streams and republishes them with a polling and/or push-based (Websockets, MQTT) interface. These streams can be consumed by other RDF stream processors (RSP) for continuous query answering with SPARQL-based query models (C-SPARQL [15], CQLS [16], TPF Query Streamer [17]). Dell'Aglio et al. [18] describes for the publication of an RDF stream that both push-based and polling interfaces can be supported; the consumer may choose what it prefers. Also, several requirements [19] are defined for RSP query engines of which requirement 6 "timely fashion" acknowledges [18] that the timing of results depend on the application scenario and thus the requirements of the consumer of the data publication or query service. In the next section, we will look into how the timely fashion requirement is applied for live Open datasets.

3 Field report on live Open datasets

This section gives an overview of how many live Open datasets are available in the European and U.S. Open Data portals, what the rate of publication is and which update mechanism is used. Datasets were retrieved by doing a full-text search on "real-time". Only working and up-to-date datasets are mentioned. Note that this is a non-exhaustive overview, because among other reasons not all Open datasets are harvested by these portals. We also briefly describe the update mechanisms that are used in the public transport and cryptocurrency trading domains.

Table 1: Overview of live Open datasets according to their country, how fast it updates and whether a polling or push interface is used.

Country	Datasets	Update interval	Update mechanism
Belgium	Vehicles position (Public transport MIVB)	20s	Polling
Belgium	Bicycle counter	realtime	Polling
Belgium	Park+rides	realtime	Polling
France	Parking and bicycle stations availability	60s	Polling
Sweden	Notifications about Lightning Strikes	realtime	Push-based
Ireland	Weather station information, the Irish National Tide Gauge Network	3600s	Polling
UK	River level data	900s	Polling
UK	Cycle hire availability & arrival predictions (Transport for London Unified API)	300s	Polling
UK	Arrival predictions (Transport for London Unified API)	realtime	Push-based
U.S.	Real-Time Traffic Incident Reports of Austin-Travis County	300s	Polling
U.S.	True Time API (arrival information and location of public transport vehicles)	realtime	Polling
U.S.	Current Bike Availability by Station (Nextbike)	300s	Polling
U.S.	USGS Streamflow Stations	24h	Polling
U.S.	NOAA water level (tidal) data of 205 Stations for the Coastal United States and Other Non-U.S. Sites	360s	Polling
U.S.	National Renewable Energy Laboratory [20]	60s	Polling
U.S.	RTC MetStation real time data	360s	Polling
U.S.	Seattle Real Time Fire 911 Calls	300s	Polling

On Table 1 we see that live Open datasets from only five countries are harvested by the European Open Data portal. While we can argue that there are more relevant datasets than on this overview, for example by browsing for Open Data portals of specific cities or using Google Dataset Search [21], we still get a broad view of the current state-of-the-art interfaces. Only 2 datasets have a push interface available, both using the Websocket protocol, of which only the Transport for London (TfL) API from the U.K. is free to use. Interestingly, the True Time API from the U.S. offers the same functionality as the TfL API with arrival predictions for public transport, but uses polling instead of push-based update mechanism. On the one hand, we found live datasets related to the environment (water level, weather, etc.) that publish at a lower rate (from every minute to every hour) with a polling-based interface. On the other hand, we found mobility related datasets whose update interval fits between realtime (as fast as possible) and 5 minutes.

We also examined the public transport domain where GTFS-RT, a data specification for publishing live transit updates, takes no position [22] on how updates should be published, except that HTTP should be used. OpenTripPlanner (OTP), a world-wide used multi-modal route planner which allows retrieving GTFS-RT updates and bicycle availabilities, also supports both approaches: setup a frequency in seconds for polling or subscribe to a push-based API.

When people want to trade money or digital coins, it is crucial that the latency of price tickings, books, etc. are as low as possible, otherwise it can literally cost them money. The rate of publication of these live datasets are also typically below 1 second. Therefore, publishers in the cryptocurrency trading domain heavily use push-based mechanisms for their clients. This however does not mean that a HTTP polling approach is not used. Websockets are de-facto used as a bidirectional communication channel is required for trading. We tested three publishers (Bitfinex, Bitmex and gdax) and saw over a span of one year that they were still available which makes us believe that Open push-based interfaces are a viable option for other domains as well.

4 Problem Statement

In the field report, we saw that there is no strict guideline whether to use polling or pushing for a certain dataset. Based on the insights from the field report and related work, we define the following research question:

Research question: Which kind of Web interface for server to client communication is best suited for publishing live Open Data in function of server-side cost, scalability and latency on the client?

Following hypotheses are defined which will be answered in the discussion:

H1: Using a Server-Sent Events interface will result in a lower latency on the client, compared to a polling interface.
H2: The server-side CPU cost of an HTTP polling interface is initially higher than a Server-Sent Events interface, but increases less steeply when the number of clients increases.
H3: From a certain number of clients onward, the server cost of a Server-Sent Events interface exceeds the server cost of a polling interface.

5 Benchmark HTTP polling versus Server-Sent Events

5.1 Evaluation design

Update interval of live dataset. The experiments focus on observing the latency on the client when a server needs to serve a high number of clients. As we want to observe the latency on the client per update and we expect a higher latency when the server

works under a high load of clients, it is important to reserve enough time between updates. Table 1 shows that most datasets have an update interval in the range of seconds. By choosing a fixed update interval of 5 seconds for the live dataset in this experiment, there should be enough time to observe the latency on the client between two updates and still have a representative update interval according to Table 1. Also, an invariantly changing dataset allows to set HTTP caching headers according to the update interval, which is an opportune circumstance for HTTP polling.

Live dataset. A JSON object with a size of 5.2 kB is generated every 5s, which has a similar size as the park+rides dataset (5.6 kB) from Table 1. This object is annotated with a timestamp that indicates when this object was generated and is used by clients to calculate the latency on the client. Furthermore, it is published inside a HTTP document for clients that use HTTP polling or it is directly pushed to clients with SSE.

Latency on the client. The goal of this benchmark is to observe the time between the generation of an update and when a client can further process it. We define this as the latency on the client of an update. For HTTP polling, this depends on timing its request as closely as possible after a new update is available. Based on the caching headers of a response (Cache-Control for HTTP/1.1 or Expires for HTTP/1.0), a client could calculate the optimal time for its next request. In this benchmark, we choose to continuously fetch the HTTP document with a pause of 500ms between the previous response and the next request, because we expect a similar polling implementation by Open Data reusers like OpenTripPlanner.

Web API. The live data is published with a server written in the Node.js Web application framework Express and exposes 2 API routes: /polling to retrieve a JSON document containing the latest value with an HTTP GET request and /sse to receive updates through an open TCP connection with Server-Sent-Events. The latter is naively implemented server-side with a for loop that pushes updates to every client. Multiple optimizations are possible (multi-threading, load balancing, etc.), but to make a fair comparison between HTTP polling and SSE we focus on having a single-threaded implementation for both approaches. By only using a for loop, all work needs to be done by the default Node.js single-threaded event loop. For HTTP polling, we use nginx as reverse proxy and enable single threading by configuring the number of worker_processes to 1. In order that nginx can handle many simultaneous connections with clients, the number of worker_connections is set to 10k.

HTTP caching. Two HTTP caching components are available: one is implemented server-side using the HTTP cache of nginx, the other one at the client-side. When a client fetches the document containing the most recent update, it will first check if a non-expired copy is available in its cache (Fig. 1). Web browsers have this feature enabled by default, but the Node.js clients in this benchmark need to use the cacheable-request NPM package to support HTTP caching. An unexpected side effect of using nginx (version 1.17.7) is that it does not dynamically update the max-age value in the Cache-Control header when returning a copy from its cache. This means that a cached copy with a time-to-live of 1s will still have a max-age of 5s which leads into extra client-side caching for 5s instead of 1s. To circumvent this behaviour in our benchmark,

we also added the Expires header, which indicates when the document is expired. This requires that the clocks of the server and clients are synchronized which is the case for the testbed we used. For future work, we suggest to use Varnish as reverse proxy, which dynamically updates the Cache-Control header. In Fig. 1, we see that the client makes a request to nginx when its cache is expired. When nginx's cache is also expired, then only the first request will be let through (proxy_cache_lock on) to retrieve the document from the back-end server over a persistent keep-alive connection that is configured. Other requests need to wait until nginx received the response and then pull from the updated cache. The max-age value is calculated by subtracting the time that is already passed (the current time - the time the last update is generated) from the frequency a new update is generated (5000ms). The Expires header is calculated by adding the update frequency to the time of the last update. Finally, nginx removes the Cache-Control header, which obliges the client to use the Expires header for the correct timing of its cache.

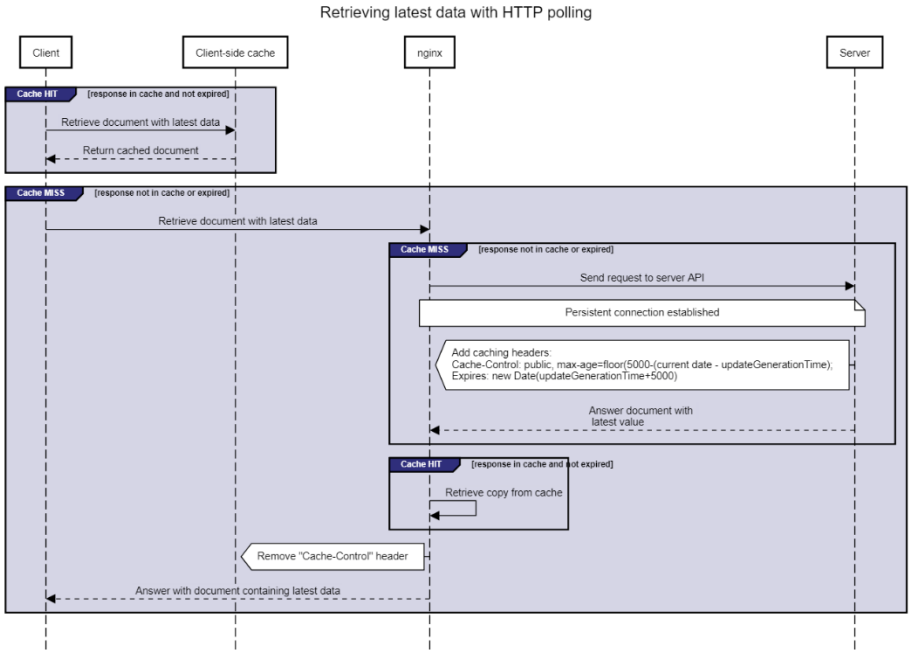

Fig. 1. Two cache components (client-side and server-side/nginx) are used for HTTP polling. As Nginx does not dynamically update the max-age value from the Cache-Control header, we fall back on the Expires header for client-side caching.

High number of clients. A benchmark environment is created using the Cloudlab testbed at the University of Utah, which had at the time of writing the biggest number (200+) among alternatives available to us of bare metal servers. This is necessary for our envisaged scenario where we need to deploy thousands of Web clients to simulate the impact on an Open Data interface. 200 HP ProLiant m400 [23] servers are used for our benchmark, each containing a CPU architecture with eight 64-bit ARMv8 (Atlas/A57) cores at 2.4 GHz, 64 GB of RAM, 120 TB of SATA flash storage. Notice that

a m400 server uses ARM which is generally lower in performance than traditional x86 server architectures which could lead to faster detection of performance losses. Lastly, we use the Kubernetes framework to easily orchestrate the deployment and scaling of our server and clients that are containerized with Docker.

Logging results. A time series database (InfluxDB) is deployed where clients log their latency on the client. Also, the visualisation tool Grafana is deployed to monitor whether all clients are initialized and polling or subscribed as expected and then to export the results as CSV. When an update is received on the client, only 10% is randomly logged to InfluxDB to prevent an excess of updates. We exported several minutes of recordings of the latency on the client per test, which we deem enough, for evaluation. To log the usage of the server and client (CPU and memory), we use the Kubernetes Metrics Server. Similarly to retrieving the resource usage of a Linux machine with the 'top' command, we can use 'kubectl top pods' to extract the resource usage of Kubernetes pods. For each test, we ran this multiple times and calculated the average for plotting. The Node.js back-end and nginx reverse proxy are deployed in one single pod. This means we can easily monitor the overall resource usage for HTTP polling from both components together.

5.2 Results

The results from our benchmark are split into two parts: first, we will show the latency on the client with density charts. Then we will look into the resource usage. We will first test HTTP polling without using nginx. This way, we demonstrate the performance boost nginx creates.

Polling without nginx. On Fig. 2, we can see a group of density charts for polling without using nginx. The y-axis represents the number of clients (100, 1000, etc.) that are deployed in polling mode, while the x-axis represents the time in ms it took to retrieve a new update. For every number of clients, there is a separate density chart showing the distribution of latencies on the client that are measured. A client still uses a client-side cache and polls every 500ms, but it directly contacts the server Web API when its cache expires. For 100 clients, more than half of the updates is retrieved below 0.5s. Up to 2000 clients, the majority of updates are retrieved beneath one second. Above 2000 clients, the server struggles to respond efficiently as the latency on the client is spread from 0s up to 5s. We were unable to deploy more than 5000 clients, because the server fails to handle the number of requests.

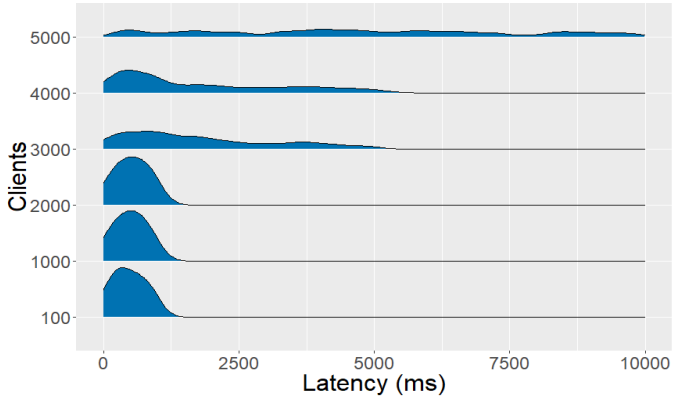

Fig. 2. Latency on the client with polling without using nginx. The server is able to answer effectively up to 2000 clients and becomes unstable above 5000 clients.

Polling. With nginx added to the server as a reverse proxy with HTTP caching enabled, we can see on Fig. 3.1 that the server is able to handle 8000 clients instead of 5000 clients and have a similar latency for 100 and 1000 clients as without nginx (Fig. 2). From 2000 clients on, a peak of the latency on the client appears between 1s and 2s. Also, a peak exists between 3s and 4s starting from 4000 clients. At 8000 clients, the distribution is evenly spread between 0s and 2s. Clients have a polling frequency of 500ms and start polling at different times, which is one of the causes of this spread. In addition, all requests wait until the cache is updated and then nginx returns responses single-threaded. To see whether this is not caused by our client implementation, we performed a benchmark with the wrk HTTP benchmarking tool, which generates a significant amount of requests to test the HTTP response latency instead of the latency on the client to retrieve an update. A wrk benchmark was performed for 30 seconds, using 12 threads, keeping 400 HTTP connections open and timeout for response times above 4s. Wrk measured a maximum response latency of 3.89s with 13 responses timed out above 4s and could reach 2.52k requests/s which acknowledges insights from the density charts on Fig. 3.1.

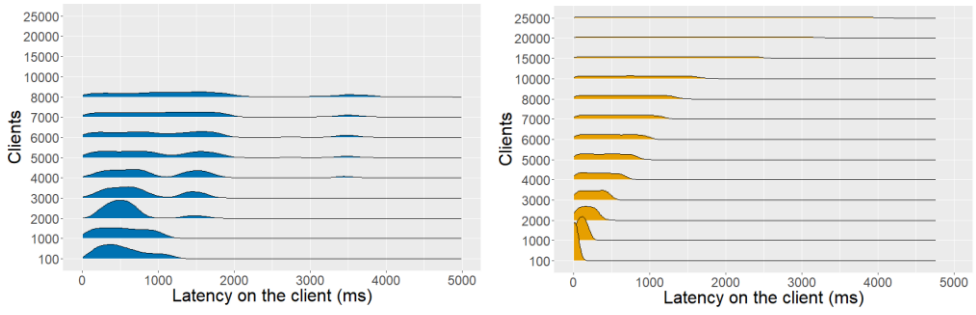

Fig. 3. Latency on the client with polling (Fig. 3.1) and Server-Sent Events (Fig. 3.2). Polling scales up to 8k clients, while Server-Sent Events can serve 25k clients.

Server-Sent Events. On Fig. 3.2 we can see that the maximal latency on the client with a SSE interface increases with the number of clients. For 5k clients, the latency on the client is still below a second, but for 25k clients this is evenly distributed between 0s and 4s. During implementation, we faced a kernel buffer issue where data transmission is queued until no data was written from Node.js to the HTTP response objects from clients. This caused an increasing minimal latency on the client and also a higher maximal latency on the client up to 1.5s for 20k clients. A continuous data transmission was achieved by running a sleep function of 1ms per 1000 clients, because we saw on the density charts of Fig 3.2 that SSE could respond efficiently up to 1000 clients.

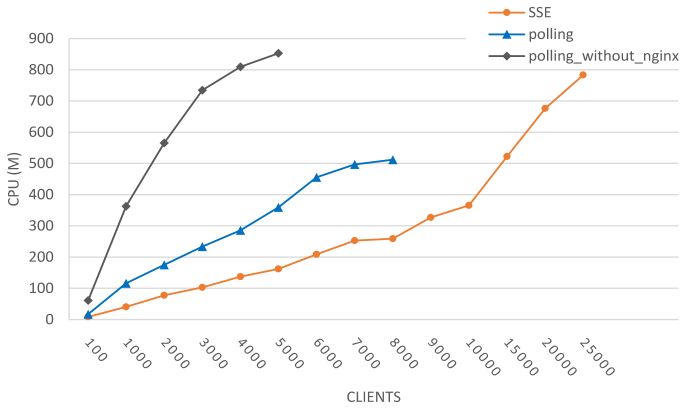

Fig. 4. CPU usage (in milliCPU) of polling and Server-Sent Events. SSE uses less milliCPU than polling.

Resource usage. We measured the CPU and memory usage for the three above-mentioned approaches. The CPU metric of a Kubernetes pod, in which our server resides, is measured in mCPUs (milliCPUs). 1000 mCPUs are equivalent to 1 AWS vCPU or 1 Hyperthread on a bare-metal Intel processor with Hyperthreading. Memory usage is measured in mebibytes (MiB). On fig. 4, we can see that polling without nginx has the steepest curve for CPU and that SSE still has a significant CPU advantage over polling with nginx. We believe this is caused by SSE having less overhead than polling, although nginx is able to minimize this with connections that are kept alive, gzip compression and caching. For 5000 clients, we see that nginx decreases CPU usage by half compared to polling without nginx. The memory usage stabilizes for polling (Fig. 5) with preference for polling with nginx. For SSE, this continuously increases with the number of clients as every client connection is held in memory.

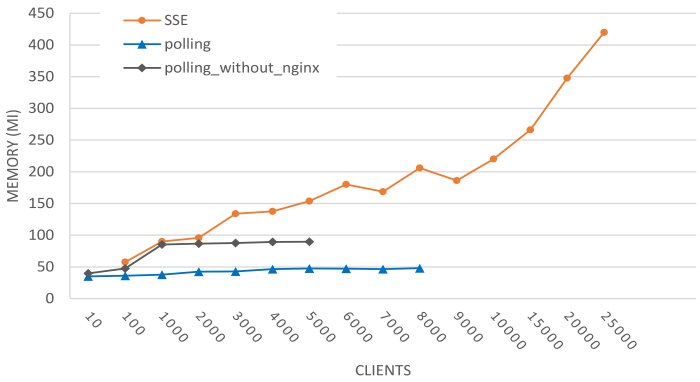

Fig. 5. Memory usage (in mebibytes) of polling and Server-Sent Events. Polling has a low memory footprint (<100 MiB), while SSE needs to keep every client connection continuously in memory**.**

6 Discussion

Based on the benchmarking results from previous section, we start our discussion by verifying our hypotheses. Thereafter, we will answer our research questions more in depth.

Our first hypothesis H1 states that a SSE interface will result in a lower latency on the client than with polling. When comparing the latency the client on Fig. 3, we see that SSE always has a lower maximal latency on the client than a polling interface with server-side cache enabled, so we accept H1. Surprisingly, to achieve this distribution for SSE from 0s onwards, we had to add a sleep function of 1ms between every 1000 clients. Otherwise, all responses are first stored in the kernel buffer before getting transmitted which causes a higher minimum latency on the client. This behavior is not described in implementation guidelines of SSE or Websockets so we hope that this article can help informing the Node.js community.

For our second hypothesis H2, we expected a faster growing server cost of SSE than polling. Previously [7], it was argued that the capabilities of HTTP caching would outperform the server cost of SSE for a high number of clients, although polling has an initial higher server cost. CPU usage results (Fig. 4) show that nginx indeed improves the CPU usage for polling, but it is still steeper than SSE, so we reject H2. Publishers should take note of the higher memory footprint of SSE, because all client connections are saved in memory. When data needs to be encrypted using TLS, then we expect that the server-side CPU cost for both polling and pushing will be only be slightly higher, because TCP connections are kept alive for both approaches and thus the time and resource expensive TLS handshake only needs to be done once for the first HTTP request. [24].

Hypothesis H3 can also be falsified, because the CPU usage of polling grows apart higher than SSE for a large number of clients. In other words, answering requests with

a cached copy still has a higher CPU cost than pushing updates directly. In terms of scalability, we saw in Fig. 3 that SSE is able to serve 25k clients, while polling could only serve 8k clients. From our hypotheses, we see that SSE is favored based on scalability, server cost and latency on the client.

The field report (Section 3) shows for the vehicle positions dataset that updates are generated every 20s and polling is used. When an end user wants to reuse this information inside the OpenTripPlanner application, then a polling frequency (s) needs to be specified. This depends on the maximal latency on the client that an end user perceives as acceptable. Even if the update interval of the live dataset is known, e.g. every 5s like in our benchmark, matching the polling frequency with this update interval would still create a latency on the client distribution between 0s and the polling frequency. In the worst case, an HTTP response is returned just before a new update arrives, so the client will only fetch this update with the next request round. Given the results of our hypotheses, we question at which point the maximum acceptable latency on the client (MAL) of an end user must be in order that our polling interface can serve more clients than SSE. In the next paragraph, we will describe how we can theoretically calculate this MAL cut-off point between polling and SSE.

In our benchmark, we tested with the Wrk HTTP benchmarking tool that our polling interface could serve up to 2.52k requests/s. When all users would have started polling at different starting times every 5s, so they configured a MAL of 5s, then our polling interface could have served theoretically up to 12.6k clients (2.52*5) instead of 8k from our benchmark. This can be generalized with the following formula, which states that the maximum supported number of requests/s of a polling interface (2.52k in our case) must be higher than the expected number of users that make one request every MAL seconds:

$$Requests_{maxsupported}/s \geq Users * Request/MAL \quad (1)$$

Open Data publishers can calculate with formula (1) how many users with a certain MAL can be maximally served. In practice, users can configure a higher polling frequency than the expected MAL, so the number of users that can be served will probably be lower. To compare this with SSE, we see on Fig. 3.2 that SSE can maximally serve 25k clients over a MAL of 4s, which is still more than polling can serve (10k clients) with this MAL. By increasing the MAL, we found that a MAL of 10s allows our polling interface to serve the same number of clients (25k) as SSE. This should not be interpreted as a universal number, because there are other factors that can influence this number: an AMD CPU architecture could be more performant than the ARM architecture we used or publishing in a global network could add extra network latency for polling [5]. Nonetheless, this number gives an indication that users must have a relatively high MAL (> 10s) in order that polling can serve a higher number of clients than pushing with the same amount of resources. This brings us to our research question where our single-threaded comparison shows that pushing is the best choice up to a maximum latency on the client of 10s. For datasets where all users configure a MAL above 10s, then a polling interface is capable to serve a higher number of clients, which can be calculated with formula (1).

Fig. 4 showed that HTTP caching is crucial for a polling interface to lower the CPU usage, but for variantly updating datasets it is not possible to foresee when the next

update will happen. For this use case, we advise to only cache the response for 1s in the reverse proxy (micro caching), and thus still off-load the back-end Web API. At last, caching headers should always be set if possible, according to the arrival of the next update so the user can still configure its preferred MAL and bandwidth usage can be reduced.

7 Conclusion

In related work of the RDF Stream Processing Community Group [18] and the field report on live Open datasets (Section 3), we saw that publishing live changing resources on the Web leaves the options for polling and push-based mechanisms open. With this article, we shed some light into this topic by running a benchmark between a Server-Sent Event and polling interface. In contrast with traditional HTTP benchmarks, we focused on assessing the latency on the client of an update instead of the HTTP response latency. We extend existing work [5], because we saw that a push mechanism is also the best option when the server needs to handle a high number of clients. If the latency on the client must be as low as possible, then the server CPU cost of HTTP polling with caching enabled does not outperform pushing [7]. Data publishers can use our results, which reflect the performance of a pull and push mechanism over a single thread, to foresee when to scale their infrastructure in function of the number of clients and the expected maximum latency on the client. The application scenario that users have a maximal acceptable latency on the client (MAL) of at least 10s makes polling more scalable than pushing, although this is a theoretical number. Configuring the MAL is a task that an Open Data reuser should be able to choose and this cannot be forced by the data publisher by setting a caching header. Because of this, caching headers should always be applied for invariant streams, but its timing should not be further than the next update. For variantly updating streams where the timing of the next update is unknown, we advise to use micro caching on the reverse proxy. At last, Open Data publishers should do user research (conduct a survey or investigate query logs [25]) to find out which MAL is most likely to be used for each dataset and verify if their current infrastructure is fit for this by applying formula (1). For example, the vehicle position dataset from the field report in Section 3 has a new update available every 20s. If users also configure their polling frequency in function of this interval, so their MAL is above 10s, then polling is the preferred interface based on the number of clients that can be served.

The “timely fashion” requirement is currently only applied for each component individually (from Web stream to RDF stream and RSP query engines). In future work, we would like to investigate how this requirement can be resolved from a true user perspective, such as Smart City Dashboards, and how this requirement goes top-down to all the underlying components.

8 Acknowledgements

We would like to express our gratitude to Raf Buyle and Pieter Bonte for their support during the writing process of this article.

9 References

1. Berners-Lee, T.: 5 Star Data. https://5stardata.info/en/ (2009), last accessed 2020/03/04.
2. Colpaert, P., Van Compernolle, M., De Vocht, L., Dimou, A., Vander Sande, M., Mechant, P., Verborgh, R., Mannens, E.: Quantifying the Interoperability of Open Government Datasets. Computer. 47, 50–56 (2014).
3. Rezaei, R., Chiew, T.K., Lee, S.P.: A review on E-business Interoperability Frameworks. Journal of Systems and Software. 93, 199–216 (2014).
4. Lubbers, P., Greco, F.: Html5 websocket: A quantum leap in scalability for the web (2010). URL http://www. websocket. org/quantum. html. (2016) , last accessed 2020/03/04.
5. Pimentel, V., Nickerson, B.G.: Communicating and Displaying Real-Time Data with WebSocket. IEEE Internet Computing. 16, 45–53 (2012).
6. Słodziak, W., Nowak, Z.: Performance Analysis of Web Systems Based on XMLHttpRequest, Server-Sent Events and WebSocket. In: Information Systems Architecture and Technology: Proceedings of 36th International Conference on Information Systems Architecture and Technology–ISAT 2015–Part II. pp. 71–83. Springer (2016).
7. Rojas Meléndez, J.A., Van de Vyvere, B., Gevaert, A., Taelman, R., Colpaert, P., Verborgh, R.: A Preliminary Open Data Publishing Strategy for Live Data in Flanders. In: Companion Proceedings of the The Web Conference 2018. pp. 1847–1853. International World Wide Web Conferences Steering Committee (2018).
8. Colpaert, P., Verborgh, R., Mannens, E.: Public Transit Route Planning through Lightweight Linked Data Interfaces. In: Cabot, J., Virgilio, R. de, and Torlone, R. (eds.) Proceedings of the 17th International Conference on Web Engineering. pp. 403–411. Springer (2017).
9. Naik, N.: Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In: 2017 IEEE International Systems Engineering Symposium (ISSE). pp. 1–7 (2017).
10. Genestoux, J., Fitzpatrick, B., Slatkin, B., Atkins, M.: WebSub W3C Recommendation 23 January 2018. https://www.w3.org/TR/websub/, last accessed 2020/03/04.
11. Taelman, R., Van Herwegen, J., Vander Sande, M., Verborgh, R.: Comunica: a Modular SPARQL Query Engine for the Web. In: Vrandečić, D., Bontcheva, K., Suárez-Figueroa, M.C., Presutti, V., Celino, I., Sabou, M., Kaffee, L.-A., and Simperl, E. (eds.) Proceedings of the 17th International Semantic Web Conference. pp. 239–255. Springer (2018).
12. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal. 15, 121–142 (2006).
13. Dell'Aglio, D., Della Valle, E., Calbimonte, J.-P., Corcho, O.: RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems. International Journal on Semantic Web and Information Systems (IJSWIS). 10, 17–44 (2014).
14. Mauri, A., Calbimonte, J.-P., Dell'Aglio, D., Balduini, M., Brambilla, M., Della Valle, E., Aberer, K.: Triplewave: Spreading RDF streams on the web. In: International Semantic Web Conference. pp. 140–149. Springer (2016).
15. Barbieri, D.F., Braga, D., Ceri, S., VALLE, E.M.A.N.U.E.L.E.D.E.L.L.A., Grossniklaus, M.: C-SPARQL: a continuous query language for RDF data streams. International Journal of Semantic Computing. 4, 3–25 (2010).
16. Le-Phuoc, D., Dao-Tran, M., Parreira, J.X., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: International Semantic Web Conference. pp. 370–388. Springer (2011).
17. Taelman, R., Verborgh, R., Colpaert, P., Mannens, E.: Continuous Client-side Query Evaluation over Dynamic Linked Data. In: Sack, H., Rizzo, G., Steinmetz, N., Mladenić, D.,

Auer, S., and Lange, C. (eds.) Proceedings of the 13th Extended Semantic Web Conference: Satellite events. pp. 273–289. Springer (2016).
18. Dell'Aglio, D., Phuoc, D.L., Le-Tuan, A., Ali, M.I., Calbimonte, J.-P.: On a Web of Data Streams. DeSemWeb@ISWC. (2017).
19. Dell'Aglio, D., Della Valle, E., van Harmelen, F., Bernstein, A.: Stream reasoning: A survey and outlook. Data Science. 1, 59–83 (2017).
20. Jager, A., D.; Andreas: NREL National Wind Technology Center (NWTC): M2 Tower (1996).
21. Brickley, D., Burgess, M., Noy, N.: Google Dataset Search: Building a Search Engine for Datasets in an Open Web Ecosystem. In: The World Wide Web Conference. pp. 1365–1375. Association for Computing Machinery, New York, NY, USA (2019).
22. Google: GTFS Realtime Overview. https://developers.google.com/transit/gtfs-realtime, last accessed 2020/03/04.
23. Hardware HP ProLiant m400 server at Cloudlab Utah. https://docs.cloudlab.us/hardware.html, last accessed 2020/03/04.
24. Nginx SSL/TLS offloading. https://www.nginx.com/blog/nginx-ssl/, last accessed 2020/03/04.
25. Vandewiele, G., Colpaert, P., Janssens, O., Van Herwegen, J., Verborgh, R., Mannens, E., Ongenae, F., De Turck, F.: Predicting train occupancies based on query logs and external data sources. In: Proceedings of the 7th International Workshop on Location and the Web (2017).