uc3m | Universidad Carlos III de Madrid

e-Archivo

This is a postprint version of the following published document:

García-Reinoso, Jaime; Fernández, Norberto; Vidal, Iván; Arias Fisteus, Jesús (2015). Scalable Data Replication in Content-Centric Networking based on Alias Names. *Journal of Network and Computer Applications*, (2015), v. 47, pp.: 85-98.

DOI: https://doi.org/10.1016/j.jnca.2014.10.003

Ztreamy: A middleware for publishing semantic streams on the Web

Jesus Arias Fisteus[a,1], Norberto Fernandez Garcia[a], Luis Sanchez Fernandez[a], Damaris Fuentes-Lorenzo[a]

[a]*Depto. de Ingeniería Telemática, Universidad Carlos III de Madrid, Avda. de la Universidad, 30, 28911, Leganés (Madrid), Spain*

Abstract

In order to make the semantic sensor Web a reality, middleware for efficiently publishing semantically-annotated data streams on the Web is needed. Such middleware should be designed to allow third parties to reuse and mash-up data coming from streams. These third parties should even be able to publish their own value-added streams derived from other streams and static data. In this work we present Ztreamy, a scalable middleware platform for the distribution of semantic data streams through HTTP. The platform provides an API for both publishing and consuming streams, as well as built-in filtering services based on data semantics. A key contribution of our proposal with respect to other related systems in the state of the art is its scalability. Our experiments with Ztreamy show that a single server is able, in some configurations, to publish a real-time stream to up to 40000 simultaneous clients with delivery delays of just a few seconds, largely outperforming other systems in the state of the art.

Key words: semantic sensor Web, stream distribution middleware, semantic stream

1. Introduction

Nowadays, there are well known best practices for publishing linked data on the Web in an interoperable manner, so that it can be retrieved, queried, browsed, etc. by applications [1]. However, they are principally aimed at publishing static data, and do not properly accommodate the vast amounts of time-dependent data about the physical world produced by sensors. This kind of dynamic data is not restricted to physical sensors. For example, there are *social sensors* that capture information published in the social sphere (social networks, blogs, etc.)

Data from sensors is usually processed in the form of streams. In [2] a data stream is defined as *a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items.* Streams are different to stored data in several aspects: they cannot normally be stored in their entirety, and the order in which data is received cannot be controlled. Wrapping sensor data into semantic formats like RDF, and extending it with semantic annotations, facilitate data integration. In the same way the linked data initiative does, publishing in an open manner on the Web this dynamic information, accompanied by proper semantic annotations, opens the door to independently-created applications and mashups. These applications can exploit information in unforeseeable ways

by integrating sensor data, linking to sources of static information, etc. This kind of platform is called the *semantic sensor Web* [3]. Current research in this area is aimed at providing solutions to problems such as the annotation and transformation of data coming from sensors, integration of data from heterogeneous models, integration of sensor data with linked data (and static data in general), discovery of relevant streams, querying and reasoning on streams, provenance of data (e.g. identifying the quality or reliability of different streams, sensors, etc.), large scale distribution of streams, etc.

In this article we present Ztreamy, a middleware for publishing streams of semantically-annotated data on the Web. By using it, data sources can publish their streams, and applications can consume them. It also allows operations such as mirroring, joining, splitting and filtering these streams. There are other frameworks that can be used for publishing sensor data, such as DataTurbine [4]. Some have even been designed for RDF sensor streams, such as the Linked Stream Middleware (LSM) [5]. Our most important contributions with respect to previous work are the scalability of the proposal and its use of HTTP, which makes it available to a wide range of application environments. As we show in the evaluation, Ztreamy can handle much bigger data rates and number of clients from a single server than other existing solutions. In addition, it provides mechanisms for broadcasting the streams from several servers when additional performance is needed. Another difference with respect to some of the available solutions is that it provides built-in services for manipulating data represented with the RDF data model. This reduces the effort needed to develop applications on top of it, because of the widespread availability of tools (query proces-

Email addresses: `jaf@it.uc3m.es` (Jesus Arias Fisteus), `berto@it.uc3m.es` (Norberto Fernandez Garcia), `luiss@it.uc3m.es` (Luis Sanchez Fernandez), `dfuentes@it.uc3m.es` (Damaris Fuentes-Lorenzo)

[1]Corresponding author's phone number: +34916245940; Fax: +34916248749

sors, data stores, reasoners, etc.) for RDF.

The rest of this paper is organized as follows. Section 2 describes a scenario that motivates the need of our proposal and its main requirements. Section 3 describes the design decisions behind Ztreamy. We evaluate its performance and compare it to other systems in section 4. Section 5 discusses other existing middleware platforms for sensor networks. Finally, section 6 concludes this article.

2. Motivation and requirements

In this section we introduce a motivating scenario based on the concept of *smart cities*, in which sensor networks play a key role. The SmartSantander project [6] is one of several examples of research initiatives in this area. They have in common that a large number of physical sensors (traffic, weather, air quality, noise, etc.) are deployed across the city. They may be complemented by other kinds of dynamic data sources, such as the current status of public transport services, the schedule of traffic lights for the next few minutes, tweets geo-located in that city, data contributed by citizens through applications installed on their smartphones, etc. We are especially interested in an open scenario that enables an ecosystem of data providers and third-party applications that offer meaningful services to citizens by using that data. In this scenario, providers (e.g. public entities and companies) publish dynamic data in the form of streams. We identify some requirements that a platform for publishing streams in such an scenario should fulfill:

1. The ability for applications to consume the streams of data they are interested in. For example, a traffic application does not probably need to know about the status of the water supply systems, and a neighborhood-specific application may not need to subscribe to streams from other parts of the city. Providers should be able to logically group data into separate streams. Client applications would subscribe just to the streams they are interested in.

2. The ability for any party to publish value-added streams by merging, splitting, filtering, processing, enriching with external data, etc. other streams, or just mirroring them for a better load distribution. This would foster the openness of the platform and contribute to the appearance of innovative services. For example, a third party could publish a new stream that enriched the current traffic status stream with predictions based on statistics about the past and foreseeable circumstances such as planned cultural events and demonstrations, bad weather, etc.

3. The ability to scale with the number of consumers and the data rate. A big city would produce vast volumes of data and support large amounts of consumers simultaneously subscribed to the most popular streams. A system that does not scale can greatly degrade user experience or even be unfeasible. Apart

from being scalable, the system should also be cheap enough to enable budget-limited parties (small companies, nonprofits, individuals) to publish their own small-scale streams from consumer-grade devices and network connections.

4. Accessibility from a wide range of application environments. The use of a wide-spread standard protocol such as HTTP would ease the creation of applications for desktops, smartphones, Web browsers, etc. in almost any programming language. Providing middleware libraries for the most common ones would also contribute to this objective.

5. Configurable quality of service, such as delivery reliability and latency. Firstly, consumers should not miss current items in the stream. In case of a network disruption, they should be made aware and, if the disruption is reasonably short, be able to retrieve the items they missed as soon as they reconnect. Secondly, the platform has to distribute data as quickly as possible, because responsiveness is a key factor in some applications such as public emergency announcements. We believe that maximum delays of a few seconds under normal network conditions are a sensible target to achieve. Publishers should be able to configure, according to their needs and the characteristics of the stream, the size of the time window for which they are able to redeliver data to clients that suffer disruptions, and the latencies they target, because both parameters have an associated cost.

6. Flexibility in network layout. Publishers should be able to deploy their infrastructure according to the requirements of the application, from simple client-server layouts to complex tree-like distribution. This aspect is also related to the scalability of the system. For example, publishers might want to replicate from each area of the city the streams that are popular there.

7. Semantic annotation. We believe that wrapping data with semantic annotations, as promoted by the semantic sensor web [3], is necessary for data interoperability and integration in such a vast and heterogeneous scenario. The platform should support its use and provide facilities based on those semantics. For example, an application could use semantic techniques to select relevant data from a stream and enrich it with static linked data.

3. The Ztreamy stream distribution platform

Our main objective is making Ztreamy a scalable middleware platform for publishing and consuming semantically annotated data streams on the Web, for scenarios such as the one presented in the previous section. In this section we explain and justify how we have devised it from the points of view of architecture, data representation and data transport. Further implementation details are available at [7].
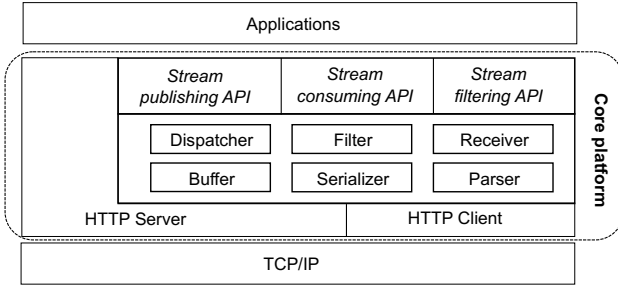
2

Figure 1: Architecture of Ztreamy.

3.1. Architecture

Following the definition in [2], we consider that a stream is a sequence of timestamped items, being an item the basic unit of data generated by sensors (or data sources in general).

According to requirement 4, Ztreamy acts as a middleware library for applications. Figure 1 shows its architecture. The main functions that the platform exposes to applications are: consuming, publishing and filtering streams, and sending items for publication in a stream.

Also following requirement 4, items are consumed, published and sent to streams through HTTP. This allows access from every major application environment, even for applications that do not use our platform as a library. When there are security requirements, such as confidentiality, integrity or access control, HTTPS can be used instead. Consumers subscribe to a stream by just sending a GET HTTP request to its URI. The URI also allows them to select the access mechanism: long-lived requests or long-polling. For scalability reasons, long-lived requests, also known as HTTP streaming, are the preferred mechanism. With this mechanism the server sends the HTTP response in chunks as new data is available, but never finishes the response neither closes the connection. This way of communication is efficient because just one underlying TCP connection is used and just one HTTP request needs to be processed for a possibly long period of time.

The long-lived requests mechanism is complemented with buffering for improving performance. Instead of sending items to consumers as soon as they are available for publication, they are stored in a buffer, which is periodically dumped to the network. As shown in the experiments of section 4, using this buffer may significantly improve performance: A bigger buffer window (i.e. a bigger period) reduces CPU consumption, improves compression ratios and helps to reduce network load (requirement 3). Its downside is that it delays the delivery of data, although this increase in delay is predictable and bounded by the period at which the contents of the buffer are sent. The platform allows that period to be configured, or the mechanism to be disabled if needed.

Some HTTP client libraries are not compatible with long-lived requests, as they wait the response to be finished before returning data to the application. In that case, consumers can use long-polling instead, in which the server finishes the response as soon as all the available data has been sent, and the consumer sends a new HTTP request immediately after that. Servers need to keep the most recent data items in memory, so that long-polling clients can receive them in their next request. This feature is also used for providing reliability to long-lived clients (requirement 5), because they are able, in case of network disruptions, to receive the missed items after reconnection, if they are still in memory. The publisher can configure how many items are stored in memory for this purpose, and therefore control the period of time for which a disruption may happen without the client losing data.

The use of HTTP has also the advantage of allowing conventional load balancing techniques for scalability (requirement 3), such as those based on DNS or front-end reverse proxy servers.

Filtering allows applications to select the part of a stream that matches some criteria (requirement 1). Filters can be installed at the server side by the publisher, e.g. for automatically splitting one stream into more specific streams (requirement 2), and at the client side by the consumer application. Some filters exploit the semantic annotations of the data (requirement 7). At the moment, Ztreamy already provides some types of built-in filters:

- *By source or application*: Given the identifier of a specific data source (e.g. a specific sensor) or application, the filter selects the items from them.

- *By vocabulary*: Given a URI prefix (e.g. a namespace URI), it selects the items whose triples contain URIs with the given prefix.

- *By SPARQL queries*: Given an ASK SPARQL query, it selects the items for which the query evaluates to true. This filter works on the graph of a single item of the stream.

- *Custom filters*: In case more complex filtering behaviors are needed, the platform allows applications to register their custom filter components, which implement a simple interface. The platform runs those filters once for every data item. However, because they can keep memory, they are also able to base their decisions on previously seen items.

On top of the basic API provided by the core of the platform, other necessary services for the semantic sensor Web could be built, although they are out of the scope of our contribution. For example, continuous query systems can be developed with the custom filters facility.

Ztreamy is flexible in how nodes are laid out in the network. It consists of interconnected nodes, which may act with one or more of these basic roles:

- Sources: They produce original data. In the case of physical sensors with limited computing capacity, or not directly connected to the network, the source may be the computer that reads or adapts their data.

- Publishers: They are HTTP servers that publish one or more streams. They receive the data to be published from sources or other streams.

- Consumers: They subscribe to the streams they are interested in.

This flexibility helps to comply with requirements 1, 2, 3 and 6. For example, a node acting as both a consumer and a publisher can repeat streams for load balancing, join several streams into one, filter a stream or split it into several ones by means of filters, produce higher-level streams by doing some processing on other streams, etc. It is also possible for a node to integrate the source and publisher in the same process.

When servers apply buffering, cascading several of them (e.g. cascaded repeaters) makes data be buffered once per server, which increases end-to-end delivery delays. In order to prevent this, the platform provides for each stream a special URI to access an unbuffered version of it with priority delivery, intended to be used in those cases.

3.2. Data representation and compression

Data items in a stream are represented with headers and body. Headers contain some metadata about the item, whereas the body contains its main data. Mandatory headers include a unique item identifier, a creation timestamp, the identifier of the source of the item and the data type and length of its body. Optional headers include application identifier and identifiers of the intermediate nodes the item was transmitted through. Applications may also use custom extension headers. According to requirement 7, the body of the item should normally be a serialization of RDF, such as Turtle. Figure 2 shows an example. Ztreamy provides code for the creation, serialization and parsing of RDF data. Non-RDF data can also be transported, e.g. in the first stages of its acquisition, but in order to exploit the benefits of semantic technologies it should be semantically annotated somewhere in its processing pipeline.

Experiments with a preliminary prototype showed that the bandwidth in the server is one of the main limiting factors to the number of simultaneous clients and data rates it can handle. Because data is represented with textual serializations of RDF, and it is usual for sources to produce regular structures, there is room for big gains by compressing the data. However, due to the probably small size of the items in the stream, compressing each item in isolation would not achieve significant compression ratios. Therefore, stream compression techniques should be used. We chose the general-purpose Zlib library[2], because it is widely available, compatible with the standard Deflate protocol and able to work in stream mode. Nevertheless, compressing streams instead of single items poses a challenge: in order to reduce CPU and memory consumption

```
Event-Id: 1100254f-f4ba-49aa-8c47-605e3110169e
Source-Id: 83a4c888-c395-4bb7-a635-c5b864d6bd06
Syntax: text/turtle
Application-Id: identi.ca dataset
Timestamp: 2012-10-25T13:31:24+02:00
Body-Length: 843

@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix geo: <http://www.w3.org/2003/01/geo/wgs84_pos#> .
@prefix webtlab: <http://webtlab.it.uc3m.es/ns/> .
<http://identi.ca/notice/97535534>
    dc:creator <http://identi.ca/user/94360>;
    dc:date "2012-10-25T11:28:51+00:00";
    webtlab:content
        "Completed registrations for #wmbangalore !Wikimedia
        DevCamp Banglalore: 2430 applications,
        130 invitations sent http://is.gd/FtXMhT";
    webtlab:conversation
        <http://identi.ca/conversation/96703048>;
    webtlab:hashtag "wmbangalore";
    webtlab:location [ a geo:Place;
        geo:lat "13.018",
        geo:long "77.568" ] .
<http://identi.ca/user/94360> foaf:based_near [ a geo:Place;
        geo:lat "52.392";
        geo:long "4.899" ];
    foaf:name "S....... M......" .
```

Figure 2: Example of a post retrieved from the Identica microblogging service and wrapped with semantic metadata. It includes the headers that our platform introduces.

in the server, the stream should ideally be compressed only once for all the consumers. However, new subscribers to an already running stream would need to know the current compression context in order to begin to decompress the stream. In order to address this issue, we use a main compression context, common for all the consumers except the recently connected ones. New consumers are served, at the beginning, data compressed in an ad-hoc compression context. Periodically, the state of the main context is reset so that those new consumers can synchronize to it and abandon their ad-hoc context. The main context is reset periodically, instead of whenever there is a new subscription, because there is a cost in compression ratios every time its state is reset, due to the compressor losing its memory about previously seen data.

General purpose or RDF-specific compression algorithms would have been another alternative for compression. An RDF compression system that achieves better compression ratios than general purpose algorithms is presented in [8]. However, it has been designed for compressing large RDF graphs instead of streams.

3.3. Implementation of Ztreamy

We have implemented Ztreamy on top of the Tornado Web server[3]. We selected it because of its efficiency when handling large amounts of long-lived clients, as well as the convenient asynchronous server and client HTTP libraries it provides. The efficiency of Tornado is due to its use of non-blocking input/output (the *epoll* notification services of the Linux kernel, and similar services in other

---

[2]http://www.zlib.net/ (Available 9th Jul. 2013)

[3]http://www.tornadoweb.org/ (Available 17th Jan. 2013)

platforms), which allows the system to efficiently handle many network sockets from a single thread, instead of the traditional approach of using blocking input/output in one thread per client. It avoids, when there is a large number of clients, the cost associated to the management by the operating system of such a number of threads.

We have publicly released our prototype of Ztreamy with an open source license[4]. It provides application programming interfaces so that stream clients and sources can be developed in the Python programming language with a minimal effort. Nevertheless, as they communicate with servers through HTTP, clients and sources can be developed in any other language, including JavaScript executed from Web browsers. This prototype implements all the functions of the core platform depicted in figure 1.

4. Performance evaluation

We carried out a series of experiments with Ztreamy and other systems in order to measure how their performance evolves as the number of clients connected to the stream and the data rate change. We used the following performance indicators:

- CPU use: absolute amount of CPU time the server needed to process a given load. Because the experiments were designed so that a source sends data for an average duration of $100s$, the percentage of a core of the CPU used by the server can be approximated by this CPU time, unless it enters saturation.

- Delivery delay: time from the instant the source is ready to send an item to the instant the client application parses it.

The experiments were run with the stream servers of all the systems we compare installed on a Linux 3.0.2 computer with an Intel Core i7 CPU at 2.8 GHz, with 4 cores and hyper-threading, and 8 GB of RAM memory. In order to get a high number of clients connected to the server, clients were run from 4 different computers, up to 10000 clients per computer. Some clients were located in the same 1 Gbps Ethernet network as the server, while others were in a different 100 Mbps LAN. Round-trip transmission time within the LAN of the server was around $0.1ms$, and around $0.5ms$ to the second LAN. Servers and clients were run on CPython 2.7 virtual machines. In order to precisely measure delivery delays, all the machines were synchronized with the same NTP server. Their estimated offsets with respect to the reference were checked in every experiment to be less than $5ms$.

The timing of the items in the stream followed a Poisson process with a data rate that varied depending on the experiment, and an average duration of $100s$. Each experiment was repeated 10 times with the same parameters.

We computed the 95% confidence interval for each data point. However, since they were negligible, and in order to improve legibility, we do not show those confidence intervals in the plots.

The dataset consisted of posts captured from the microblogging site Identica. We have published this dataset along with all the software and instructions needed to repeat our experiments[5]. We converted each Identica post into an item in the stream, represented using Turtle. The dataset contains 25749 posts published in a period of 4 days. The average size of a post, once it is represented with Turtle, is 880.95 bytes.

The results we report are specific for this dataset and could change for others. The characteristics of the dataset that affect results the most are the size of the items and compressibility. Other factors such as the hardware of the server and the performance of the network have also some impact. However, we are less interested in absolute performance values than in comparing the systems under the same conditions and studying how performance varies as different parameters change (number of clients, data rate).

4.1. Experiments with a single server

In this section we analyze the performance of Ztreamy and other systems when run in a single server. We selected for comparison[6]: Dataturbine [4], for it being a leading free software for sensor data delivery; Faye[7], a widely used free software HTTP publish-subscribe framework; ZeroMQ[8], the messaging system that Storm (an open-source distributed stream processing software maintained by Twitter) uses to distribute its streams through the cluster, in publish-subscribe communication mode; and the websockets-based publishing module of LSM [5] on top of the Apache Tomcat server. In the case of LSM, instead of directly using their code, we adapted it in order to avoid using the query engine, so that their results are not penalized for being the only platform running a query engine. We were not able to compare proprietary systems such as Xively or the infrastructure behind the streaming API of Twitter.

Ztreamy was run in four configurations (with and without buffering, with and without compression). In this section we just report the results of our best configuration ($0.5s$ server buffer and compression). The other three variants are discussed in section 4.3, where we explain the reasons behind the performance of our system.

First, we served a 4 $items/s$ stream to a variable amount of clients. We run each system with a growing number of clients until its performance degraded too much (excessive delivery delays, data loss or malfunctioning). Figures 3a, 3c and 4a show the results. Dataturbine and Faye stopped working properly at 700 clients, the former with a 3% data loss and high delays, and the latter with frequent crashes

---

[4]http://www.it.uc3m.es/jaf/ztreamy/ (Available 17th Jan. 2013)

[5]http://www.it.uc3m.es/jaf/ztreamy/doc/experiments/
[6]Dataturbine 3.2, Faye 0.8.6, ZeroMQ 2.2.0 and LSM 1.0.0
[7]http://faye.jcoglan.com/ (Available Jul. 11th 2013)
[8]http://www.zeromq.org/ (Available 9th Jul. 2013)

beyond that number of clients. LSM begun suffering from high delays beyond 2000 clients and losing data beyond 5000. ZeroMQ performed better than the other systems until approximately 8000 clients. At that point, it saturated the network and became unstable. At 12000 it was unable to deliver about 8% of the items one minute after the source finished sending data. Ztreamy, because of it compressing the data, did not suffer from this. At 40000 clients it still delivered the stream with less than $3s$ median delay and around 90% of CPU use (this can be seen in the "1 server" curves of figure 5).

Then, we run the systems with 500 clients and a variable data rate. Figures 3b, 3d and 4b show the results. Dataturbine reached a 2.5% data loss at about 30 $items/s$. Faye delivered all the data even at its saturation point, but suffered from massive delays. LSM did not lose data in this experiment, but begun having high delays beyond 25 $items/s$. ZeroMQ begun to drop data at high rates, with a 6% data loss at 80 $items/s$, due to bandwidth saturation. Ztreamy performed stably for these rates. The reason it performed better than ZeroMQ in terms of CPU use, on the contrary to the previous experiment, is that the $0.5s$ buffer is more effective at higher rates because it aggregates more data in each period. Although not shown in the plot in order to avoid compressing the x axis too much, Ztreamy maintained its performance until approximately 250 $items/s$. At that point, it delivered data with $5s$ delay and less than 25% of CPU use, but bandwidth begun to affect its performance.

4.2. Load balancing with repeaters

We repeated the same experiment (4 $items/s$ and a variable number of clients), but adding two more servers that acted as repeaters for the stream. The three servers were run with a $0.5s$ buffer, although the repeaters connected to the master server through the unbuffered channel, as explained at the end of section 3.1. Clients were evenly balanced so that each server handled 1/3 of them. Figure 5 compares delays and CPU consumption when serving the stream with the two repeaters and without them. With small amounts of clients, delays are similar in both deployments. However, the deployment with two repeaters is able to keep delays stable and lower for larger amounts of clients, as expected because the use of CPU in each server is smaller since they handle less clients each.

4.3. Discussion

Our conclusion from the comparison with other tools is that Ztreamy is able to handle many more clients and much higher data rates. Equally to our proposal, all the other systems transmit on top of TCP and maintain the connection open. Therefore, we can discard this as a reason for the improvements. Buffering in the server and compression are, however, not implemented in any of the other systems that we analyzed.

The performance of the other systems, except ZeroMQ, began to drop when their CPU use reached the limit. The server window buffer mechanism presented in section 3.1 is clearly important to reduce CPU use, as it can be seen by comparing our prototype with and without buffering in figures 3a and 3b. As expected, the data also shows that the more items the buffer aggregates, the bigger the effectiveness of buffering. This happens when the data rate increases (figure 3b) and when the period of the buffer increases (figure 6). The latter shows the big impact that different buffer sizes have in CPU use in a situation in which not using buffering would result in an extreme saturation of the server.

Our data shows the effect of buffering in delivery delays. Since there is a compromise between buffer size and latency, implementing an adaptive buffer window size instead of a fixed-length one would be useful for maintaining delays as small as possible for the current server load.

The experiments also show that compression, as presented in section 3.2, contributes to the performance of Ztreamy. It reduced application-level traffic to a 15%. This is its key advantage with respect to ZeroMQ, which performed quite well until it congested the network. When no server buffer is used, compression has no significant effect. This can be observed in all the plots, in which the two curves with no buffering almost overlap. The effect of compression is neither important when the buffer is so small that it aggregates small amounts of data (figures 3a and 3c), until the network begins to saturate. At that point our system with $0.5s$ buffer but no compression began to lose data. Figure 3b shows that, however, at higher data rates, at which the buffer aggregates more data, compression gives significant gains in terms of CPU. The time needed to compress data is compensated by the time saved in the networking stack because of transmitting less data. Therefore, compression, when combined with a big enough buffer, not only reduces network traffic but also saves CPU use, which allows the system to handle bigger loads.

The underlying servers of Ztreamy, ZeroMQ and Faye use single-threaded non-blocking input/output, which we explained in section 3.3. Despite Faye not performing well, we believe that it has an important contribution in the performance of our prototype and ZeroMQ. However, we cannot measure the gains it is providing to Ztreamy, because it is inherent to it and cannot be disabled.

Another conclusion that can be drawn from this data is that if ZeroMQ implemented buffering and compression it would probably outperform our system. This is so because it does not implement an HTTP server and its code is native, while Tornado is completely written in Python, with no native code. Note that, however, compression is not trivial to implement in publish-subscribe communication patterns, due to new subscriptions arriving at any time, as explained in section 3.2.

Results also suggest that in Ztreamy data rates have less effect in CPU use than the number of clients. Transmitting higher rates to less clients tends to saturate the network before the CPU of the server, whereas transmitting lower rates to more clients tends to saturate server
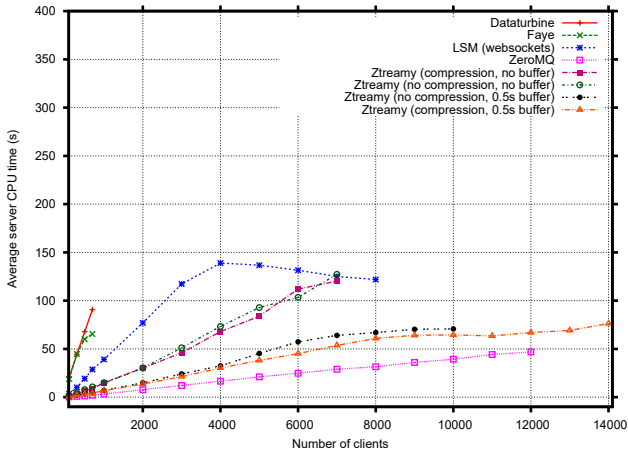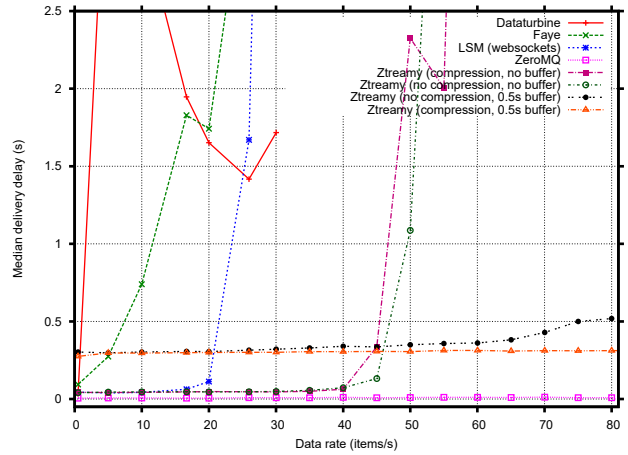
(a) CPU time needed to send 400 items at average 4 *items/s* to a variable number of simultaneous clients.
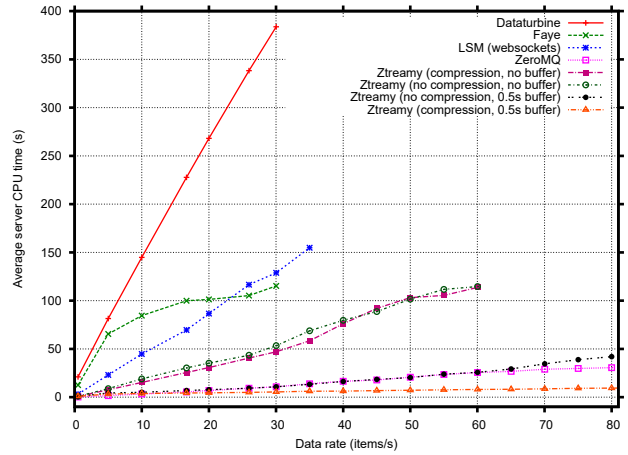
(b) CPU time needed to send items at different rates to 500 simultaneous clients.
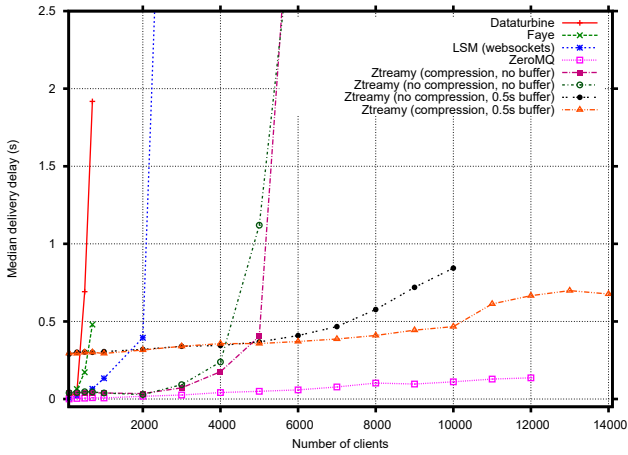
(c) Delivery delays when sending 400 items at average 4 *items/s* to a variable number of simultaneous clients.

(d) Delivery delays when sending items at different rates to 500 simultaneous clients.

Figure 3: Compared CPU use and delivery delays for all the systems.

CPU before the network.

Finally, section 4.2 showed that the facilities for repeating streams improve the performance of the system in situations in which a single server would be in saturation.

5. Related work

Xively[9] (formerly COSM and Pachube) is a commercial service to which data consumers and producers connect to exchange real-time data. To the best of our knowledge, there is no public information about the engineering of their infrastructure.

Systems such as Global Sensor Networks (GSN) [9], DataTurbine [4] and BRTT Antelope[10] are well known platforms for gathering data from sensors. They support storing, publishing on top of a variety of protocols, remotely querying, processing and visualizing the data. Environmental monitoring is their most important area of application. Of these systems, DataTurbine is probably the one that provides functions that are more similar to our proposal. More specifically, it can send and receive streams of data on top of TCP and WebDAV. Although DataTurbine is a more mature and complete product, our proposal has two key advantages regarding the publishing of streams: a much better performance (see section 4) and integrated support for semantically-annotated data (e.g. SPARQL-based filtering as explained in section 3.1).

The Open Geospatial Consortium[11] (OGC) produces standards and best practices in the area of the sensor Web. Its proposed architecture is described in [10]. Regarding the access to measurements from sensors, they propose the Sensor Observations Service (SOS) and the Web Notification Service (WNS) [11]. The SOS provides a way for
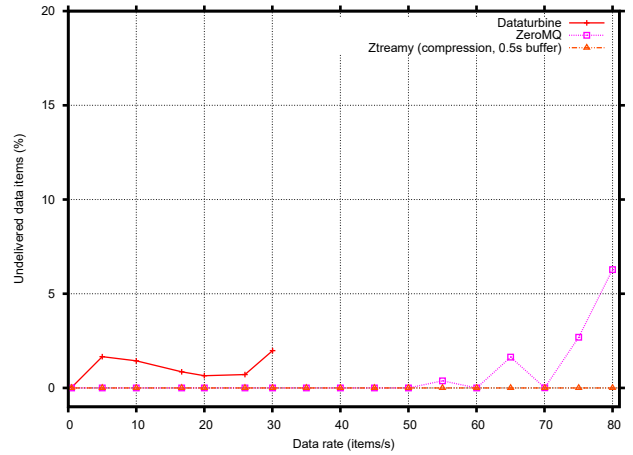
[9]https://xively.com/ (Available 9th Jul. 2013)

[10]http://www.brtt.com (Available 17th Jan. 2013)

[11]http://www.opengeospatial.org/ (Available 17th Jan. 2013)
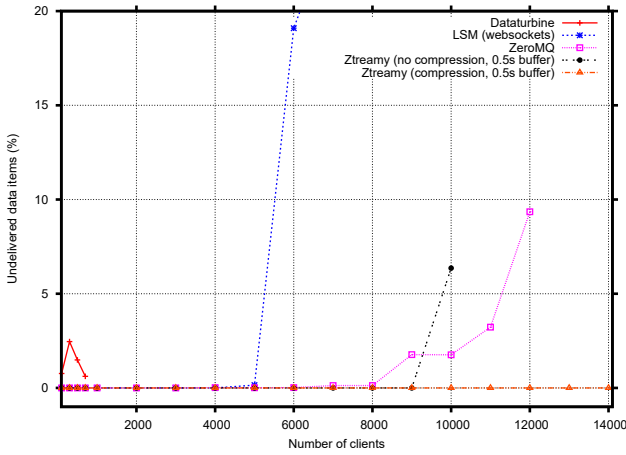
(a) Percentage of undelivered data items when sending 400 items at average 4 *items/s* to a variable number of simultaneous clients.

(b) Percentage of undelivered data items when sending items at different rates to 500 simultaneous clients.

Figure 4: Comparison of data not delivered between the start of the experiment and 60*s* after the data source sends the last event. For clarity, only the systems that lose a significant amount of data an the best configuration of Ztreamy are shown.
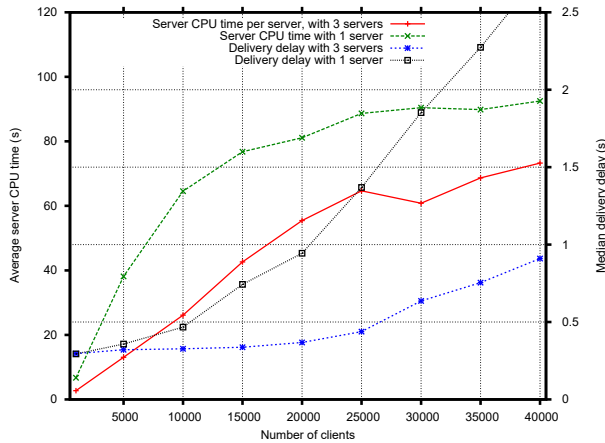

Figure 5: CPU time and delivery delays with 4 *items/s* stream in deployments with 1 server and 3 servers (1 master server and two repeaters).
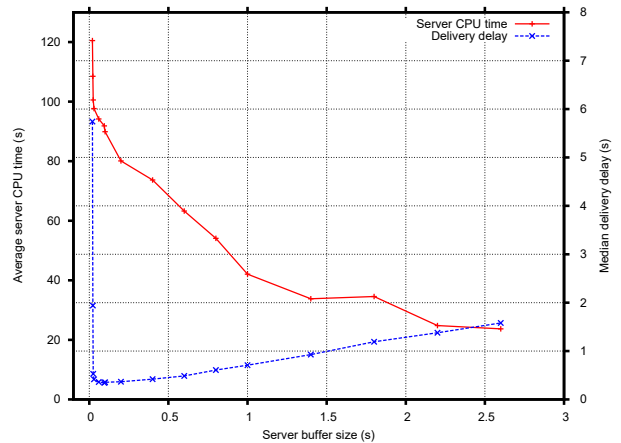

Figure 6: CPU time and delivery delays to send 2000 items at average 20 *items/s* to 8000 clients with different configurations of server buffer window.

clients to query measurements from sensors using Web services, but require clients to poll for new data. The WNS provides means to send notifications to clients, which register in the service to receive them through HTTP, XMPP or e-mail, among others. In the case of HTTP, it requires the client to provide a Web server, which can be an inconvenience in some situations such as when the client is behind a firewall or Network Address Translator (NAT). The alternatives, e-mail and XMPP, limit their usability on the Web and, in the case of e-mail, have a poor performance.

Data stream processing [12] and *complex event processing* [13] are paradigms for processing large streams of data. A complete survey on these technologies is presented in [14]. Queries in these systems run continuously, which means that they produce results gradually as new

matching data arrives. Data does not necessarily need to be stored. In many cases, especially for high data rates, it is discarded after being processed. Twitter Storm[12] is a recent but widely used open source example of such a system. Linked Stream Middleware (LSM) [5] is another example, specifically designed for processing semantically-annotated streaming data coming from sensors. Both tools provide the ability to distribute processing through a cluster. While this kind of systems focus on the infrastructure for scalably *processing* streaming data, our proposal is centered on openly *publishing* streams on the Web, so that they can be consumed by any client that has interest in them. This affects how the platform needs to be engineered, because the number of clients and their

---

[12]http://storm-project.net/ (Available 9th Jul. 2013)

connection patterns cannot normally be predicted in our scenario, whereas these aforementioned systems distribute the streams across a predictable cluster of computers usually under the control of the same administrators. Therefore, the ability of the platform to scale to large numbers of clients is more critical in our scenario. Our proposal complements those systems. They target querying and processing streams, whereas we target publishing them. In fact, our proposal could be used to feed them with streams or to publish streams resulting from their processing. Note that LSM already provides modules for publishing streams through PubSubHubbub[13] and WebSockets[14]. However, as we show in section 4, there is room for improvement in their performance.

6. Conclusions

In this paper we presented a scalable platform for publishing semantic streams on the Web. The platform is flexible, in the sense that diverse network layouts can be deployed depending on the needs of the application. Streams can be easily duplicated, aggregated and filtered. Applications written in diverse application environments can consume and publish streams by accessing platform servers through HTTP. A programming interface for Python is also available. Besides handling HTTP communication, the interface simplifies application development with functionality such as data serialization/deserialization and built-in semantic filtering.

The experiments show that our system outperforms other solutions in a single server installation. We report results in which a single server handles 4 items per second to up to 40000 simultaneous clients, which represents a throughput of $180Mb/s$ compressed application-level data, with minimal average delays, no larger than a few seconds. There are several engineering decisions behind this improved performance. Firstly, buffering data at the server side (storing data in a buffer and sending the contents of this buffer periodically) reduces CPU use in the server and lets it manage much higher loads. Secondly, although its implementation in a publish-subscribe system is not straightforward, compression not only reduces network load, which proved to be a limiting factor for some systems in our experiments, but also CPU use in the server. However, it is effective only if combined with the buffering mechanism. Finally, servers using single-threaded non-blocking input/output, such as the Tornado Web server on top of which Ztreamy is built, seem to perform better than traditional multi-threaded servers for large numbers of clients. Apart from that, the built-in facilities for replicating streams from different servers allow the system to scale when the resources of a single server are not enough.

Although Ztreamy already provides some basic semantic filtering capabilities, integrating continuous query processing and reasoning as platform services is a sensible line for future work, as it would simplify the development of applications on top of it. As these kinds of services are already being addressed by others ([15, 16, 17]), integrating some of those proposals is a possibility. Another possibility would be the integration of our platform as a publishing module in LSM [5], which already provides a scalable system for semantic stream processing, but has room for improvement in terms of how it publishes the streams. Other services, such as stream discovery, would also benefit users. This feature would be based on adding semantic metadata to sources and streams, using ontologies in the state of the art [18]. Other line of future research is adding self-organization capabilities to the platform, so that it can dynamically adapt to its operational conditions (e.g. client load). Self-organization should include mechanisms to automatically start other servers for replicating the stream when the load is high, and to redirect new and some of the already connected clients to them. We plan also to support WebSockets as an additional protocol to transmit streams, as it may be more convenient than HTTP for JavaScript clients that run in Web browsers.

References

[1] C. Bizer, T. Heath, T. Berners-Lee, Linked data-the story so far, International Journal on Semantic Web and Information Systems (IJSWIS) 5 (3) (2009) 1–22.

[2] L. Golab, M. T. Özsu, Issues in data stream management, SIGMOD Rec. 32 (2003) 5–14.

[3] A. Sheth, C. Henson, S. Sahoo, Semantic sensor web, IEEE Internet Computing 12 (4) (2008) 78 –83.

[4] S. Tilak, P. Hubbard, M. Miller, T. Fountain, The ring buffer network bus (rbnb) dataturbine streaming data middleware for environmental observing systems, in: e-Science and Grid Computing, IEEE International Conference on, 2007, pp. 125 –133.

[5] D. Le-Phuoc, H. Q. Nguyen-Mau, J. X. Parreira, M. Hauswirth, A middleware framework for scalable management of linked streams, Journal of Web Semantics 16 (2012) 42 – 51.

[6] J. Galache, J. Santana, V. Gutierrez, L. Sanchez, P. Sotres, L. Munoz, Towards experimentation-service duality within a smart city scenario, in: Wireless On-demand Network Systems and Services (WONS), 2012 9th Annual Conference on, 2012, pp. 175–181.

[7] J. A. Fisteus, N. Fernandez, L. Sanchez, D. Fuentes, Ztreamy: Implementation details, Tech. rep., Universidad Carlos III de Madrid, http://www.it.uc3m.es/jaf/papers/2013/ztreamy-report/ (2013).

[8] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, M. Arias, Binary RDF representation for publication and exchange (HDT), Journal of Web Semantics 19 (2013) 22 – 41.

[9] K. Aberer, M. Hauswirth, A. Salehi, Infrastructure for data processing in large-scale interconnected sensor networks, in: Mobile Data Management (MDM), 2007.

[10] I. Simonis, OGC sensor web enablement architecture, Open Geospatial Consortium (2008).

[11] I. Simonis, J. Echterhoff, Draft OpenGIS web notification service implementation specification, Open Geospatial Consortium (2006).

[12] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the

---

[13]https://code.google.com/p/pubsubhubbub/ (Available 9th Jul. 2013)

[14]http://dev.w3.org/html5/websockets/ (Available 9th Jul. 2013)

twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, ACM, 2002, pp. 1–16.

[13] D. Luckham, The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2001.

[14] G. Cugola, A. Margara, Processing flows of information: From data stream to complex event processing, ACM Comput. Surv. 44 (3) (2012) 15:1–15:62.

[15] D. F. Barbieri, D. Braga, S. Ceri, M. Grossniklaus, An execution environment for C-SPARQL queries, in: Proceedings of the 13th International Conference on Extending Database Technology, EDBT '10, ACM, New York, NY, USA, 2010, pp. 441–452.

[16] D. Barbieri, D. Braga, S. Ceri, E. Della Valle, M. Grossniklaus, Stream reasoning: Where we got so far, in: Proceedings of the 4th workshop on new forms of reasoning for the Semantic Web: Scalable & dynamic, 2010, pp. 1–7.

[17] J.-P. Calbimonte, O. Corcho, A. J. G. Gray, Enabling ontology-based access to streaming data sources, in: Proceedings of the 9th international semantic web conference on The semantic web - Volume Part I, ISWC'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 96–111.

[18] M. Compton, P. Barnaghi, L. Bermudez, R. García-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W. D. Kelsey, D. L. Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, K. Taylor, The SSN ontology of the W3C semantic sensor network incubator group, Journal of Web Semantics 17 (2012) 25 – 32.