Ztreamy: Implementation Details

Jesús Arias Fisteus Norberto Fernández García
Luis Sánchez Fernández Damaris Fuentes Lorenzo

October 2013

Technical Report

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

Abstract

Ztreamy is a scalable middleware framework for publishing semantic streams on the Web. This technical report complements the article *Ztreamy: A middleware for publishing semantic streams on the Web*, with design and implementation details about Ztreamy that could not be included there. More specifically, this document describes how Ztreamy represents the data of a stream, how event consumers and producers interact with a Ztreamy server and how the server itself is implemented.

Contents

1 Introduction

Ztreamy[1] is a scalable middleware framework for publishing semantic streams on the Web. The system is described in [1], including its motivation and requirements behind it, the most important design decissions and a performance evaluation in which it is compared to other existing systems. This technical report complements the information about Ztreamy in that publication by presenting the most relevant implementation details that could not be included there due to the limited available space.

A Ztreamy server is able to serve one or more streams from a single server. Streams are transmitted on top of HTTP and consist of a sequence of items represented as explained in section 2. Each stream has associated URIs that consumers can use to subscribe to them. Apart from identifying a given stream, a URI can specify other aspects such as whether compression is preferred and the access mechanism. As it is explained in section 3, consumers can use two alternative mechanism to access a stream: long-lived subscriptions or long polling. The mechanism is selected by means of the URI.

Apart from describing how clients and servers interact in the Ztreamy platform, this document presents the most important implementation aspects in section 4. Ztreamy is implemented on top of the Tornado Web server, based on a single-threaded non-blocking input-output model. Because of that, Ztreamy follows internally an event-driven architecture. Section 4 explains also how the server buffers the data in a stream and compresses it in order to improve performance.

2 Data presentation format

Ztreamy represents the data in a stream as a sequence of items. Their format is similar to the format of HTTP messages: items are represented with headers and body. We have chosen this HTTP-inspired data format instead of others because it is easy to produce and parse.

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.

The body of the item should normally be a serialization of RDF, such as Turtle. Figure 1 shows an example. Ztreamy provides code for the creation, serialization and parsing of RDF data. Non-RDF data, including binary 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.

[1]`http://www.it.uc3m.es/jaf/ztreamy/`

```
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 1: Example of a post retrieved from the Identica micro-blogging service and wrapped with semantic metadata. It includes the headers that Ztreamy introduces.

3 Client-server interaction

This section explains how consumers and data sources are expected interact with a Ztreamy server. The client APIs (Application Programming Interfaces) provided with Ztreamy already implement these interaction modes. Nevertheless, clients can be implemented in other programming languages provided that they follow these guidelines.

3.1 Consuming streams

Each Ztreamy stream has an identifying URI, which is chosen by the stream administrator. Since each stream can be accessed in different modes (e.g. compressed or uncompressed, with long-lived requests vs. long polling), the suffix of the URI identifies the access mode. For example, a hypothetical stream `http://example.com/t` would be accessed with one of the following URIs:

- `http://example.com/t/stream`: long-lived requests with uncompressed data.
- `http://example.com/t/compressed`: long-lived requests with Zlib compressed data.
- `http://example.com/t/long-polling`: long polling with uncompressed data.

In the three cases, the client subscribes to the stream by sending an HTTP request to the appropriate URI. The rest of the interaction with the server

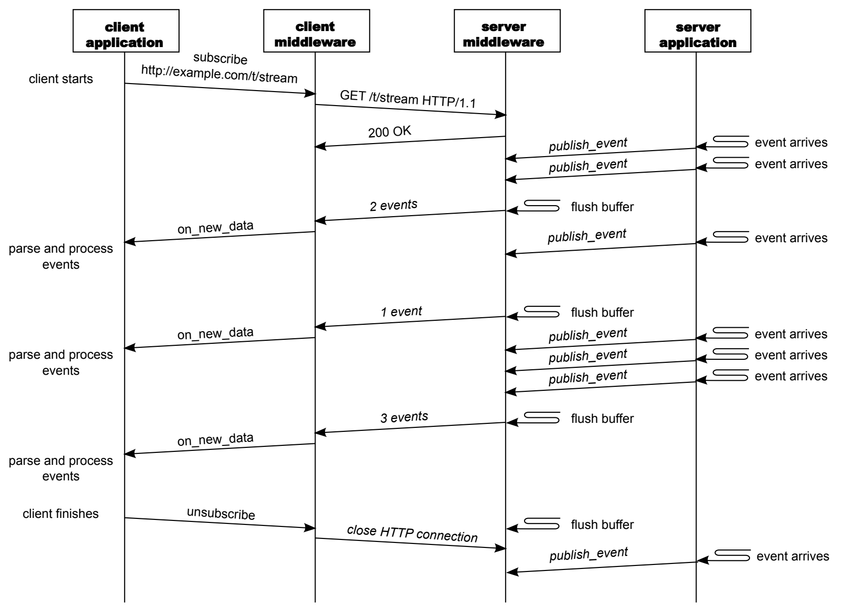

Figure 2: Interaction pattern of the long-lived requests mode.

depends on the access mode. The following sections explain the interaction pattern of each mode.

3.1.1 Long-lived subscriptions

Figure 2 shows a typical interaction with the long-lived requests mode, also known as HTTP streaming. In this mode, the client subscribes by sending an HTTP request to the server. Periodically, and only if new data is available, the server sends that data to the client, but does not finish the request. I.e., the underlying TCP connection between client and server remains open, and the client does not need to send new requests in order to get new data: the server sends it proactively. The client unsubscribes by closing the connection.

Client developers that use this mode must be aware that some HTTP client libraries block the client process until the server finishes the request. If they use a library, it must provide the ability to communicate data to the application as soon as a new chunk arrives from the server. There are appropriate HTTP client libraries available for every major programming language.

When a client subscribes to a stream, it receives only the new data that appears after the instant the subscription is established. If a client loses its connection to the server due to a network failure, an intermediate proxy that closes it or any other cause, it can ask for the events it missed. This is done by providing, in the reconnection request, the id of the last data item it received. It must me communicated as a URI parameter with name `last-seen`. For example:

```
http://example.com/t/stream?last-seen=ah34r56
```

A server receiving a request for that URI will send, before any new data, the data items that were published after the item `ah34r56` up to the instant the

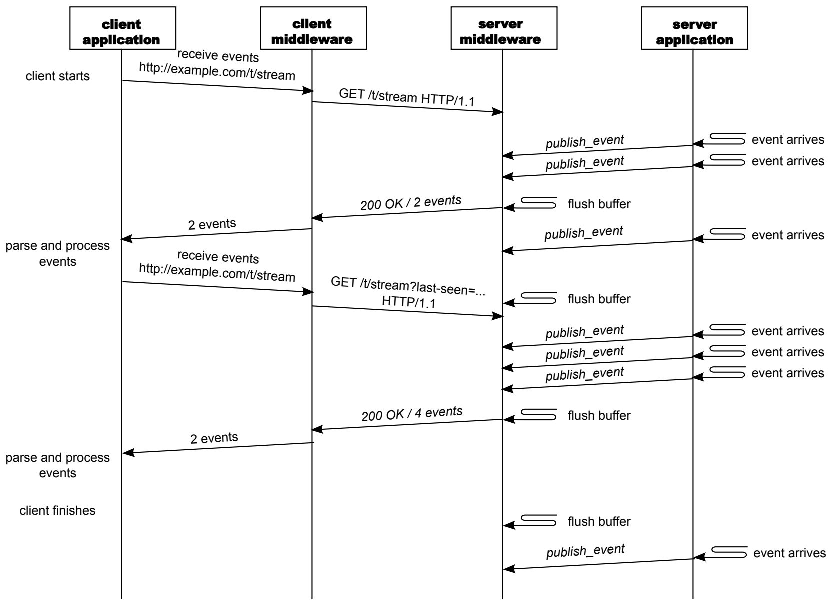

Figure 3: Interaction pattern of the long polling requests mode.

client reconnected. Since the size of the buffer in the server is bounded, data items may be lost in long disruptions. In that case, the server will not be able to recognize the event id and will send all the data it has in its buffer.

3.1.2 Long polling requests

The long-lived mode is more efficient, since just one HTTP request is sent to the server and just one TCP connection is needed. Clients that, however, do not want or cannot use the long-lived mechanism, can implement the alternative long polling mode. A typical interaction with this mode is shown in figure 3. In this mode, there is no actual subscription, in the sense that the server finishes the request each time it sends new data to the client. If the client wishes to receive more data, it needs to issue a new request each time the server finishes the previous one.

For every new request it sends, the client must specify the id of the last data item it received, so that it does not miss the data published between the end of the previous request and the processing of the new request by the server. This is done with the `last-seen` parameter, in the same way as explained for long-lived subscriptions.

The client usually sends a new request immediately after the previous request is finished. Nevertheless, the client may choose to delay it. Since the server buffers the most recent data items, data is not lost provided the period between requests is not unreasonably long. This pattern increases the delay between the publication of a data item and its reception by the client.

If a client sends a request but there is no new data available (i.e. the id of the last item it specifies corresponds to the most recent item of the stream), the server keeps the request open until new data arrives. At that moment, it sends the data and finishes the request.

Since no state is kept between consecutive long polling requests from the same client, compression is not used in this mode.

3.1.3 Compressed streams

In the long-lived requests mode, compression may be used in order to reduce network traffic. Our experiments show dramatic performance gains when compression is used. See section 4.3 for further information about how Ztreamy implements this feature.

When the client subscribes to a stream that is configured to use compression, it must process data assuming it is not compressed until the server sends a `Set-Compression` message (see section 3.3). At that moment, the client must initialize a ZLib decompressor and use it to decompress any future data it receives from that stream. Since ZLib must keep a state, the same decompressor object must be used for that stream until the client unsubscribes.

The client might receive a data chunk that includes together a `Set-Compression` message and further data. Any data that follows the `Set-Compression` message must be processed through the ZLib decompressor, even when it is received together with that command.

3.2 Sending events from producers to stream servers for publication

Data producers that wish to publish data in a stream must send the data to the stream server. Streams have a special `/publish` URI suffix that is used to send data items for publication. For example:

```
http://example.com/t/publish
```

Producers must send an HTTP POST request with that URI. The body of the request must contain one or more data items with the format specified in section 2. The HTTP *Content-Type* header must be set to:

```
application/x-ztreamy-event
```

3.3 Control messages

Ztreamy uses control messages for communicating some situations related to the stream to its consumers. Those control messages are sent as any other data item in the stream, i.e. they follow the same format as normal items. They can be separated from normal items because they use a special body syntax identifier (`ztreamy-command`). The control items that the current version of Ztreamy uses are intended to be consumed by the Ztreamy framework and not delivered to the application, although new control messages could in the future be intended for informing applications.

The control messages currently used by Ztreamy are:

- *Set-Compression*: used for communicating the consumers of a stream that the rest of the stream will be compressed with ZLib, as explained in section 3.1.3.

- *Test-Connection*: used by stream servers to check, after long periods without data in the stream, that the subscribed long-lived clients are still connected. This message can be ignored by consumers. They are not expected to answer it, because the TCP protocol automatically notifies the server about the connections that are no longer active. This message allows the server to free the memory resources associated to lost clients.

- *Stream-Finished*: used by stream servers to notify that a stream will be closed. No more items will be transmitted through the stream after this one. Consumers may disconnect after receiving it.

4 Implementation of the server

This section describes the most important design and implementation decisions that affect the behavior and performance of the server. As shown in the evaluation of Ztreamy, there are several decisions that play a big role:

- Use of single-threaded non-blocking input/output: Ztreamy works on top of the Tornado Web server, which uses the non-blocking input/output APIs available in modern operating systems. The servers of this kind tend to work better when handling large amounts of simultaneous clients than multithreaded servers that use the traditional blocking APIs.

- Buffering data at the server: Instead of sending data as soon as it is available for publication, the server can buffer the data and, periodically, send all the data in the buffer to the clients. This mechanism is optional, and its period can be configured for the needs of the application.

- Use of compression: Data can be compressed with the stream compressor ZLib. Since typical RDF data has a high degree of redundancy, it can dramatically reduce network traffic (about 85% less traffic in our experiments), which allows more clients to be handled from a single server when the available bandwidth is a limited. In addition, it may also save CPU when there is a big number of subscribed clients.

The next sections explain how Ztreamy servers implement these three mechanisms, as well as other features.

4.1 Single-threaded non-blocking input/output

The Tornado Web server implements its networking capabilities on top of non-blocking APIs of the operating system. For example, it uses in Linux the *epoll* kernel event notification mechanism instead of blocking with the system calls *select* or *poll*. The *epoll* system is more efficient when watching a large number of file descriptors[2], because it works in constant time with respect to the number of descriptors, whereas *select* and *poll* work in linear time.

Because of that, the interfaces that Tornado exposes to applications are mainly non-blocking (often called asynchronous): when an application calls a function in Tornado that involves networking, such as sending data through a

[2]Network sockets in Unix follow the same abstractions as files.

socket, the function returns immediately, without waiting for the operation to complete. Once the it is completed, an event is fired and Tornado notifies it by calling the callback function the application provided. Tornado exposes an event loop, called the *ioloop*, that dispatches the events. Once an application has been initialized, it yields control to the *ioloop*.

The fact that Tornado works in a non-blocking mode is the reason that we decided to follows an event-driven paradigm in the design of the Ztreamy server. The server is normally idle, waiting for events (e.g. a new subscription is waiting, a source sends a new item for publication, the data it sent to a client arrived correctly, a timer is triggered, etc.) When an event happens, Tornado notifies the server by calling the associated callback function. The callback contains the appropriate code for reacting to that event, which sometimes includes calling other non-blocking operations and registering new callbacks. Once the callback finishes, the *ioloop* regains control and dispatches the next event if it is already available, or waits for it to occur if not.

Tornado gives full control to Ztreamy over the HTTP request. That means that Ztreamy can keep a request on hold instead of having to respond immediately as other Web frameworks usually impose. Moreover, Ztreamy can write data progressively and keep the response open for an indefinite period of time. This feature is specially relevant for the implementation of long-lived requests.

4.2 Buffering data at the server

The server can be configured to send data to subscribers as soon as a data item is ready for publication, or buffer it in order to send it later. We have seen that the latter reduces CPU use by the server and improves the effectiveness of compression. The buffer works in the following way:

- When a new data item is available for publication, it is just stored in a buffer.
- Periodically, e.g. every $0.5s$ (the period is a configurable parameter), the server sends all the data from the buffer to every registered client and empties the buffer for the next cycle.

Its implementation is based on scheduling a periodic callback in Tornado's *ioloop*. The callback sends all the items in the buffer to the currently subscribed clients. For long-lived clients, data is sent immediately but the response is kept open. For long-polling clients, the response is closed immediately after sending the data.

Our experiments show that the bigger the period, the lower the CPU use. When the load of the server is very high, it may happen that the server is still busy when the next period is fired. In that case, the *ioloop* automatically delays it until it regains control. This increases the effective period. Although it provokes bigger data delivery delays, it has the positive side-effect that the server automatically adjusts its period according to its current load: when the load is low, it works exactly at the configured period; when the load is high, the effective period increases, which allows the server to keep working with a controlled CPU use.

Buffering has the obvious cost that it increases data delivery delays. In normal situations, the average increase of delay is half the period. With high loads, since the actual period increases as explained above, delivery delays increase. In order to avoid the accumulation of delays when several servers are chained to serve a stream, Ztreamy servers provide a special URI with suffix `/priority` that bypasses the buffer. Chained repeaters should use this URI instead of the regular one. Ztreamy keeps a list of clients connected to this special URI, and sends the items to them immediately when they are available, instead of waiting to the next period of the buffer.

4.3 Compression

Experiments with a preliminary prototype of the platform showed that the available bandwidth in the server is one of the main limiting factors to the number of simultaneous clients it can handle. Ztreamy has been especially designed for transporting sensor data represented as RDF, whose usual serializations use human-readable textual data. Furthermore, it is usual for data sources to produce data items that follow homogeneous structures, and therefore are very similar (for example, several sensors of the same type will probably use the same ontologies to describe their measurements). There is, therefore, room for reducing bandwidth consumption by compressing data. However, due to the probably small size of the data items transmitted through the platform (e.g. a single measurement from a sensor), compressing each item in isolation would not achieve significant gains. Therefore, streams need to be compressed as a whole instead of independently compressing their individual data items.

We have chosen the ZLib compressor[3] for it being the most used implementation of the widely spread Deflate stream compression protocol [2]. It is available for many programming languages and platforms, and therefore does not limit the options for developing client applications.

Compressing the stream as a whole instead of single data items poses a challenge: in order to reduce the consumption of resources in the server, data should ideally be compressed only once, independently of the number of subscribed clients. However, stream compression algorithms require clients to keep state information in order to decompress new data[4]. That state is not known to new clients that subscribe to an already running stream, which would therefore not be able to decompress the data.

Ztreamy solves the problem in the following way. Each stream has an associated compression context. Data is compressed only once for all the clients of the stream, using its compression context. When a new client connects, it cannot be served data from that context, because the client does not know it. Ztreamy solves the problem by creating a specific compression context for sending data to that client. Periodically, and only if there are new clients using ad-hoc compression contexts, the state of the context of the stream is reset. Resetting its state prevents the compressor from using past data, and therefore any client can begin to decompress the data without needing to know the previous context. The compression context is reset periodically instead of whenever a new client arrives because every time it is reset has a cost in the compression ratio, due to the compressor losing its memory about previously seen data.

[3]http://www.zlib.net/

[4]Deflate decompressors are required to keep the last 32 KB of decompressed data, because the compressor may ask the decompressor to repeat a sequence of previous data within that range.

4.4 Serving past data

As explained in section 3.1, client requests may include a `last-seen` parameter that specifies the identifier of the last event the client received. This is required for implementing long-polling interactions, in order to avoid losing the data that is published between the instants at which a connection is closed and the next request from that client arrives. In addition, the mechanism helps in long-lived requests to avoid losing data when network failures cause connections to be lost.

Clients that send the `last-seen` parameter receive all the data that has already been published in the stream and is more recent than the data item whose identifier is provided. If the item with that identifier is not in memory, it is assumed to be older than the oldest item in memory. In that case, all the data in memory is sent to the client.

The server implements this feature by storing in memory the most recent data items that were transmitted through the stream. The number of items that are stored is a configurable parameter. It should be set to a sensible value, taking into account the minimum duration for which data is expected to be recoverable and the average data rate of the stream.

4.5 Handling of client requests and subscriptions

The Ztreamy server has to process long-lived requests and long-polling requests differently.

When the server receives a long-lived request, it stores it in the list of subscribed clients. If the request includes the `last-seen`, the appropriate data is sent to the client when the request is processed. Then, every time new data is available (or when a period finishes if the server uses buffers), it is sent to all the clients in the list.

When the server receives a long-polling request that contains a `last-seen` parameter, the server sends the corresponding data and finishes the request. However, if there is no `last-seen` parameter in the request, or there is but it contains the identifier of the most recent data item, the request is stored in the list of long-polling clients. When the next data item is available for publication, the server responds to all the pending long-polling requests with that data finishes their responses. Then, the server clears that list, because the clients are expected to send a new request if they wish to continue listening to the stream. If the server is configured to use buffering, the process above happens when a period finishes instead of as soon as new data is available.

5 Conclusions

This technical report complemented the information available at [1] with some technical details regarding how clients and servers interact, the data presentation format and some notes about its implementation. Further detail can be found at the source code of Ztreamy[5].

[5] `https://github.com/jfisteus/ztreamy`

References

[1] Jesus A. Fisteus, Norberto Fernandez Garcia, Luis Sanchez Fernandez, and Damaris Fuentes-Lorenzo. Ztreamy: A middleware for publishing semantic streams on the web. *Journal of Web Semantics*, 2013. doi: 10.1016/j.websem.2013.11.002.

[2] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996.