Stream Feeds - An Abstraction for the World Wide Sensor Web

Robert Dickerson, Jiakang Lu, Jian Lu, and Kamin Whitehouse

Department of Computer Science
University of Virginia
rfdickerson,jklu,jl3aq,whitehouse@cs.virginia.edu

Abstract. RFIDs, cell phones, and sensor nodes produce streams of sensor data that help computers monitor, react to, and affect the changing status of the physical world. Our goal in this paper is to allow these data streams to be first-class citizens on the World Wide Web. We present a new Web primitive called *stream feeds* that extend traditional XML feeds such as blogs and Podcasts to accommodate the large size, high frequency, and real-time nature of sensor streams. We demonstrate that our extensions improve the scalability and efficiency over the traditional model for Web feeds such as blogs and Podcasts, particularly when feeds are being used for in-network data fusion.

1 Introduction

The *Internet of Things* (IOT) refers to the vision in which computers on the Internet are able to monitor, react to, and affect the changing status of objects in the physical world. The two major forces driving the IOT are (i) the widespread adoption of RFIDs that allow physical objects to be uniquely identified, and (ii) rapid proliferation of portable devices like cellphones and wireless sensor nodes that push computation, communication, and sensing deeper into the physical world. Both of these new technologies produce streams of data that must be stored and processed.

In this paper, we address the issue of connecting these data streams with the World Wide Web. Although the terms *Web* and *Internet* are often used interchangeably, they are quite distinct: the Internet is a set of computers connected by the IP protocol, while the Web is a set of data resources that are connected by hyperlinks. Our goal in this paper is to allow data streams to be first-class citizens on the Web: every data stream has a URL, is manipulated through the HTTP protocol, and is hyperlinked to other objects on the Web. This would allow data streams to easily interoperate with other web applications and enable crawling and indexing by search engines. In addition, streams will inherit other Web features such as authentication and encryption.

There are currently several abstractions for stream-like objects on the Web. For example, a Web *feed* is an XML document that contains a dynamically changing sequence of content items, such as blog entries or news headlines. Feeds

are transported as a single file using the *pull* mechanism provided by HTTP requests. On the other hand, multimedia streams such as Internet radio stations are transported in real-time as the content is generated, using a *push* mechanism provided by protocols like the Real Time Streaming Protocol (RTSP). This protocol allows operations like "play" and "pause" on streaming media objects.

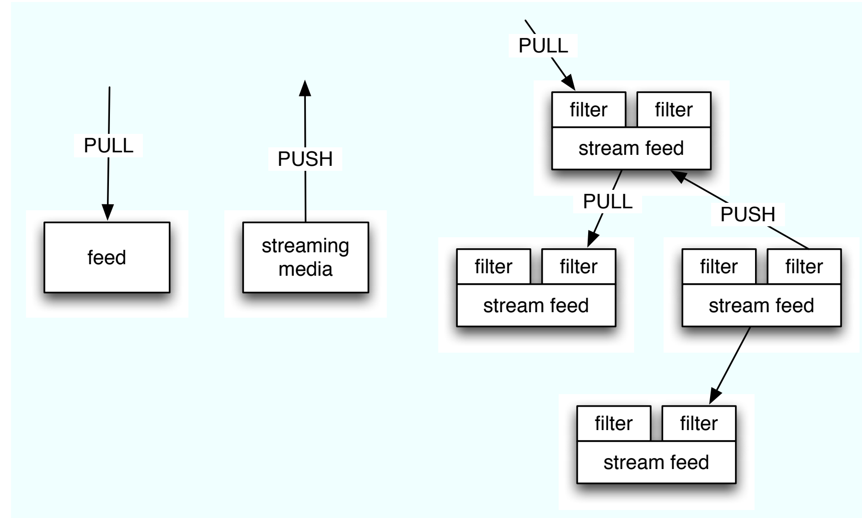

Fig. 1. Web feeds (left) only support a PULL abstraction while streaming media (middle) only supports a PUSH abstraction. The stream feed abstraction (right) must support both types of data transport, must provide server-side filtering, and must support data fusion trees.

The problem we address in this paper is that sensor streams do not fit into any of the stream abstractions currently available on the Web. Clients will sometimes want to *pull* the historical data of a sensor stream, similar to a Web feed, but other clients will want real-time updates, similar to a media stream. Furthermore, many clients will only want a small part of the total stream. Finally, unlike Web feeds or media feeds, sensor streams will commonly be *fused* to create new sensor streams.

We hypothesize that we could provide an abstraction that meets the special needs of sensor streams by adding a few simple extensions to Web feeds. To test this hypothesis, we implement a new Web abstraction called the *stream feed*. Just like normal feeds, stream feeds can be accessed through a URL and are accessed through a Web server over HTTP. As such, they can be both the source and the target of hyperlinks. They are dynamic objects that are retrieved using non-streaming transport, so updates to the content can only be observed by retrieving the entire object again. However, stream feeds also support two extensions not supported by normal feeds: server-side filtering and streaming transport. These

extensions allow stream feeds to be very large objects that change with very high frequencies and are served to clients with real-time updates.

Stream feeds can be *fused*, processed and filtered to create new stream feeds, and this process can be repeated to create a *fusion tree*. For example, data from multiple sensors on a highway can be combined to create new streams that indicate vehicle detections on the road, and these streams can in turn be fused to create a new stream with traffic conditions over time. This is different from the *aggregation* of other feeds in which, for example, news headlines from multiple sources are combined into a single feed. We demonstrate that the traditional model of serving feeds does not scale well with the number of clients and the size of the feeds, particularly with deep fusion trees, and that our stream feed extensions improve scalability, latency, and the total amount of bandwidth consumed. These results support the conclusion that stream feeds can serve as the basis for a World Wide Sensor Web in which sensor data from around the world can be shared and processed within a single global framework.

2 A Motivating Application

MetroNet is a sensor system that we are currently designing to measure foot traffic near store fronts using commodity sensors such as motion sensors and thermal cameras in downtown Charlottesville where shops are densely packed and there is high pedestrian traffic at points in the day. This simple system will be used by shops to identify the effectiveness of window displays, signs, or other techniques designed to attract foot traffic into the shop. It will measure (i) how many people walk by the shop (ii) how many people enter the shop. The ratio of these values is extremely valuable to retail stores. In addition, many stores consider the *conversion rate*, the store's number of retail transactions during a certain time period compared to its traffic, as good indicator of its performance. Measuring foot traffic can be helpful when evaluating proper labor usage, advertising and promotional programs, prime selling hours, visual merchandising strategies, future opportunities or challenges, new store concepts, store location analysis, and merchandise assortment and stock levels.

The MetroNet sensors provide valuable data to the shopkeepers, but there are also a number of compelling reasons to *share* the data. For example, shopkeepers could use data from neighboring shops to normalize their own data, allowing them to differentiate the effect of a bad storefront display from the effect of bad weather, vehicular traffic, and downtown events such as fairs or concerts. Once in place, the sensors and their data would also be extremely valuable to the residents and planners in the city. For example, potential residents that would like to buy a new house could query for aggregate activity levels in each part of the city to find a location that is close to the city's night life. This type of query could be augmented in a subsequent phase by collecting supplemental data such as average levels of noise, sunlight, humidity, or air pollutants. Potential business owners would have empirical data upon which to base the value of commercial real estate, and city planners could gear zoning and traffic decisions

to increase business and quality of life for residents. In fact, one motivation for the MetroNet testbed is to provide empirical data for an ongoing debate in Charlottesville about the effect on business of vehicular traffic crossing the downtown pedestrian zone.

Because MetroNet data can be useful to many parties, it could be used to form *fusion trees*. For example, the shopkeepers might publish streams of raw sensor data, but this sensor data may be *fused* by third parties to produce the counts of people walking in front of and into the shops. These third parties may then publish this information on their own servers as streams of people detection events. These new streams may then be fused again to produce streams with commercial real estate values or real-time estimates of overall pedestrian activity levels. A client query at the top of a tree would cause a sequence of cascading queries that consume network bandwidth and increase the latency of the query response. Stream feeds must be able to support queries over fusions trees quickly and efficiently.

There are many different requirements under which MetroNet data and the data streams that are created from it could be shared. Some data consumers, such as city planners who are moving a road to minimize effects on pedestrian traffic, may want the sensor streams as a bundle of historical data. Sometimes, the sensor stream objects could be very large (on the order of Terabytes), and so the city planners would like to query for only a small part of the entire stream, such as the data collected in the past year. Other data consumers, such as those looking at a web page with current estimates of pedestrian activity, may require real-time updates to the sensor streams. Furthermore, some data streams could be completely private, shared with other shopkeepers, or shared only with storefronts that are owned by the same company. This type of sharing may require authentication and encryption. Data could also be shared publicly, in which case the data consumer must be able to find the data through some sort of index, such as a search engine.

This case study demonstrates that stream feeds must accommodate several needs:

- *Historical queries*: many applications demand analysis over past data, and should be accessible if a server has stored it.
- *Real-time updates*: some events have low frequency but real-time importance upon firing.
- *Server-side filtering*: only a subset of data should flow from source to destinations to lessen network load.
- *Authentication*: some aggregates should be made from authorized users
- *Encryption*: some branches of the fusion tree should remain private to a certain group.
- *Indexability*: keywords from the stream meta data should be indexable along with the URL of the stream itself.
- *Scalability*: The system performance should not deteriorate dramatically as the number of clients increases, or as the depth of a fusion tree increases.
- *Autonomy*: a server should not be required to store its data on a third-party's server, and vice versa.

3 Background and Related Work

We can define a *sensor stream* to be a set of data that is generated over time by a single sensor. For example, a stream could be the set of temperature data generated by a temperature sensor over time. A stream is distinct from a *connection*, which is a software abstraction through which one device may send data to another at any time. For example, a TCP socket is a connection, but is not a stream. *Multimedia streaming* is a common technique in which a media object is transported such that it can be played before the entire file is downloaded. This is a distinct concept from a sensor stream: one property of steam feeds is that sensor streams can be transported using either streaming or non-streaming connections.

Another important distinction for this paper is the difference between the World Wide Web and the Internet. The *Web* is a set of objects that are uniquely addressed by a uniform resource locator (URL), are hosted by a *web server*, and are accessed via the HTTP protocol. Objects on the web refer to each other by their URL's, creating *links* that form a directional graph. Links in HTML documents are called *hyperlinks* while links in XML documents are called *XLinks*. The directional graph formed by these links help users to locate objects on the Web in a process called *surfing*, allow automated bots to *crawl* the Web, and form the basis for indexing and search algorithms such as Google's PageRank algorithm [1]. Although the Web is often also called the Internet, it is a distinct concept: the *Internet* is a set of computers connected by the IP protocol, while the Web is a set of objects connected by links. Most Web objects can be accessed through the Internet, but not all Internet applications are on the Web. As we describe below, most architectures that have been proposed for the World Wide Sensor Web exploit the globally-connected nature of the Internet, but do not actually put sensor streams on the Web: data streams do not have a URL, are not served by Web servers over HTTP, and cannot be the subject or object of a link. The main contribution of the stream feeds abstraction is that high data rate, aperiodic, real-time data streams become first-class citizens on the Web.

3.1 XML Schemas for Sensor Streams

The problem of sharing sensor data streams is currently becoming an active area of research. The Open Geographic Consortium (OGC) is an organization of 352 companies, government agencies, and universities collaborating to design publicly available interface specifications. The SensorNet [2] project primarily uses the OGC's schemas to achieve web interoperability. One example schema is the SensorML specification, which is an XML schema used to define processes and components associated with measurement of observations. These schemas are being created to define the a standard model for Web-based sensor networks, which will enable the discovery, exchange, processing of sensor observations, and tasking of sensor systems. However, they are not associated with a proposed architecture to store, cache, filter, and fuse data streams, and to translate between

the real-time streaming nature of data streams and the static, connectionless nature of the Web.

Other data formats include GeoRSS, which is used for encoding geographic objects in RSS fees and KML (Keyhold Markup Language), a similiar format especially designed for GoogleEarth and Google Maps [3]. However, these groups are only proposing data formats, they are not proposing an architecture or abstraction that will integrate these XML formats with the Web. Stream feeds are agnostic with respect to XML Schemas for sensor streams and any XML format may be used. Some architectures have been designed to integrate SensorML files with the Web, for example, in the Davis Weather System, AIRDAS airborne sensor, and the SPOT satellite. However, these implementations do not propose solutions to the criteria we laid out in Section 2, since they do not define complex interrelationships between data sources that are owned by several entities.

3.2 Web Feeds

A Web *feed* is an XML document that contains a dynamically changing sequence of content items, such as blog entries, Podcasts, or news headlines. Content providers *syndicate* their web feeds and content consumers *subscribe* to that feed. A subscription is usually done through a feed *aggregator*, which continually queries for the feed at regular intervals and processes it or notifies the user when the feed content has been updated. Using this model, there is no coupling between the publisher and the consumer: subscriptions can be broken by removing the feed from the user's aggregator. This model has worked well for feeds until now because traditional feeds are relatively small data objects and have little or not realtime requirements. This small size also makes it possible to filter the feed content within the client aggregator, instead of on the server side. Stream feeds differ from traditional feeds because they may often be much larger data objects, they may change with orders of magnitude higher frequency than traditional feeds, and clients that request stream feeds may require immediate update. This combination makes periodic requests for the entire data object inefficient. Another difference is that the set of operations that is typically performed on the data content of traditional feeds is limited to concatenation, sorting, and truncation of content entries. The content of sensor streams, however, can be processed, filtered, and *fused* with content from other sensor streams. There are systems that use XML files available on the web for disseminating sensor data both realtime and historical, resulting in a virtual sensor abstraction that provides a simple and uniform accesss to heterogeneous streams [4]. However, these systems do not provide a common interface for pushing and pulling for the data.

Research is interested in finding ways to search for data coming from these feeds. One example is Cobra which crawls, filters, and aggregates RSS feeds [5], and another is the popular search engine Technorati, designed to search for keywords in web feeds.

3.3 Streaming Media

The HTTP protocol was designed to retrieve an entire document at once, using a *pull* model for file transport. This model causes problems for large multimedia files that can take a very long time to download. To address this problem, several protocols and techniques have been designed to *stream* media files in order to allow the file to be played by the client before the file has been completely downloaded. For example, some systems stream data through multiple sequential HTTP responses while others use new protocols like RTP (realtime protocol) and RTSP (realtime streaming protocol) [6]. Protocols like Jabber (XMPP) [7] support streams of instant messages and can be used to stream realtime data past firewalls. The WHATWG is drafting a Web Applications 1.0 specification [8] which server-side pushing of content to the client.

Streaming file transport is used for finite-sized media files, but it also enabled the creation of *streaming media*, a type of data object that has no end, like an Internet radio station. This type of data object cannot be used with the traditional *pull* model for file transport. Sensor streams are similar to streaming media in this respect. However, the streaming transport protocols that have been designed until do not meet all of the requirements for stream feeds. They support the abstraction of a streaming media object, and provide functions such as 'play' and 'pause'. Sensor streams do not fit under this abstraction, and require other commands such as the downloading of historical data and server-side data filtering.

3.4 RFIDs

RFID tags can uniquely identify a physical object as it moves through a supply chain that may involve dozens of different vendors and shippers. For this reason, several architectures have been designed to make RFID data available from multiple administrative domains. For example, EPCglobal has created a global system for managing and sharing RFID and sensor data [9]. This system uses an asynchronous messaging protocol based on Web service standards to make Electronic Product Codes (EPC) data available in real time to hundreds of thousands of simultaneous users. However, this work differs from ours in that we are creating an abstraction based on URLs, so sensor data can be linked to by other Web objects, and can be crawled and index by search engines.

RFID applications have a *high fan-in* architecture where there are many devices at the edge of the network sending their data to a small number of more powerful processing nodes. High fan-in systems can suffer from congestion, and a system called HiFi [10] addresses this problem by performing data cleaning, event monitoring, stream correlation, and outlier detection among the mid-tier nodes.

3.5 Other World Wide Sensor Web Architectures

There have been several designs for an infrastructure for global sensor data sharing. Most architectures propose either putting all sensor data into one large

centralized database or distributing data across a distributed database running the same software stack. Centralized databases have advantages that come with simplicity, but centralization can lead to high server loads and require others to trust the server with private information. Centralization runs tangent to the web's laissez-faire philosophy. The problem with database-oriented solutions is that they force the data producer to conform to a pre-specified implementation, and do not allow for diversity of software stacks. They also do not provide rank-ordered query results so they cannot deal with issues of data fidelity and popularity.

IrisNet is a distributed database that has been proposed as an architecture for the World Wide Sensor Web [11]. Sensor data is stored in an XML database which is hierarchically partitioned and stored in different parts of the network to minimize response time and network traffic during queries. IrisNet restricts data positioning to a sensor ontology based on a hierarchy of locations. Many other systems have also been designed for distributed stream processing using distributed hash tables[12] or by moving the data to a centralized location and running continuous queries [13–15].

The SensorMap project from Microsoft Research uses a database-oriented approach where users post queries to a central database by interacting with a graphical map interface [16]. SensorMap is similar to our project in that the thrust is to make publishing data especially easy. SensorMap has explored using *screen scraping* by enabling a web crawler to automatically index adjacent text descriptions next to embedded webcam content on a webpage.

Hourglass is a system that provides circuit-based connections for real-time persistent streams [17]. It emphasizes intermittent connection scenarios for mobile or poorly connected nodes. Circuits push data through a series of links and can be processed along the way. However, because all data is pushed, Hourglass only supports real-time data streaming and does not allow searching through the history of a data stream. In addition, Hourglass does not support indexing streams and all data queries must be made through their registry/query service. Hourglass does not require a particular data schema, like database systems, but it does require users to use a particular software stack.

4 The Stream Feed Abstraction

Stream feeds provide the abstraction of a dynamic content object containing a stream of sensor data. This data object can be located with a URL, can be linked to from other data objects, and can be accessed over the HTTP protocol. For example, to access a stream of temperature data from the front door of the Whitehouse, one would visit:

```
http://www.whitehouse.gov/streams/frontdoor/temperature
```

When the client requests this URL, the client opens a TCP socket to the server and sends a HTTP GET request, as usual. The server responds immediately with all data that is already contained in the sensor stream, followed by

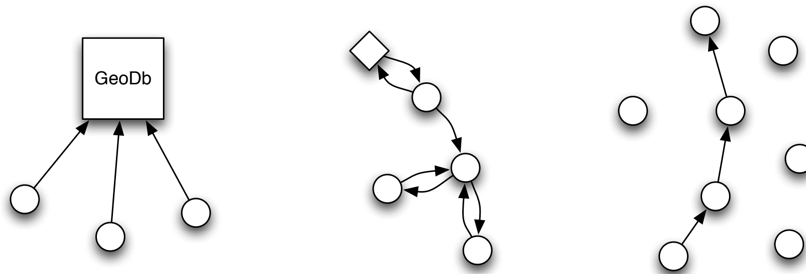

Fig. 2. SensorMap (left) collects all data into a central data base. IrisNet (center) provides a distributed XML database abstraction that can be queried by location. Hourglass (right) allows the user to setup *circuits* along which sensor data is pushed.

updates about the sensor stream in real-time as new data objects are added to the stream.

A parameter can be appended to this URI to indicate that the query should take place over a different time region. For example, the client can request:

`/streams/frontdoor/temperature/historical`

to restrict the query only to data that has already been stored, or can request

`/streams/frontdoor/temperature/incoming`

to restrict the query only to data that has not yet been received. Thus, the client can pose the query over three temporal scopes: (i) all time (default), (ii) all time before now, and (iii) all time after now. These three scopes are illustrated in Figure 3.

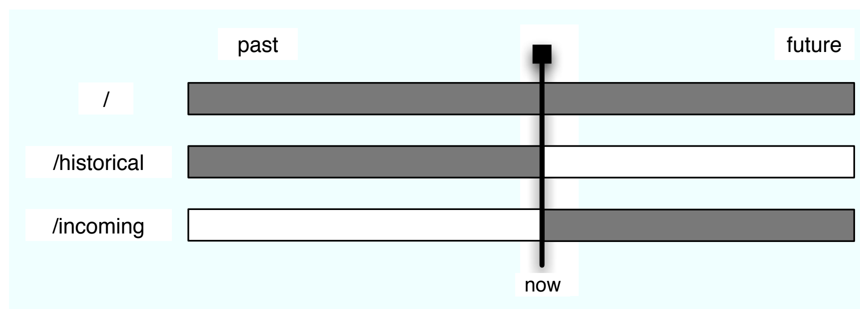

Fig. 3. Every query over a stream feed must take one of three temporal scopes. The default (top) is a query over all time, but queries over historical data (center) and incoming data (bottom) are also possible. Historical data is returned immediately while incoming data is returned as it arrives.

The user can also *filter* the sensor stream by passing filter parameters as follows:

```
/temperature/filter/?value_lower_bound=100&day_equal=Tuesday
```

This query restricts the query response to only contain the values in the data stream that have a "value" attribute greater than 100 and a "day" attribute equal to 'Tuesday'. The values that can be filtered depend on the XML schema that the sensor stream is using, so the client must be familiar with this schema. The values in the sensor stream are then filtered using syntactic filtering of the data points based on the XML tag names and values. This can be performed automatically for any XML schema. filters can be applied to a query with any of the three temporal scopes shown in Figure 3. Any tag can be used with four types of filters:

```
TAG_equal
TAG_not_equal
TAG_lower_bound
TAG_upper_bound
```

The URL-based stream feed interface conforms to REST (Representational State Transfer) principles describing how resources are defined and addressed. Unlike RPC-based web services, where a vast set of verbs to be applied to a set of nouns, the RESTful philosophy promotes the use of a simple set of verbs that operate on a vast collection of nouns. The advantage to using a RESTful interface is that there is an inherent standardization of the operations that can be applied to the resources, without needing to explicitly define descriptions of the methods that can be applied to them using the web services description language (WSDL). The URL contains all the information that is needed to return to a particular state of a web service. By including the name of the sensor, the temporal scope, and the filter parameters all in the URL, stream feeds make it easy to link to stream feeds, or even to link to partial stream feeds.

5 Implementation of the Stream Feed Abstraction

Our motivating application, MetroNet, was implemented using the Ruby on Rails framework. MetroNet is served using the lighttpd web server with the FastCGI module, and the MySQL database. The ActionController under Rails handles the complex routing requests required by StreamFeeds, so that XML feeds can be generated on demand based on the URL structure.

In MetroNet, all data is stored in a central database and a XML file is generated on the fly for each query. Thus, queries over historical data are performed as normal HTTP queries, and the generated file is returned to the client. Filters over historical queries are executed when the XML file is first generated.

Queries over incoming data are currently handled through a publish/subscribe mechanism. Every stream holds a list of subscriptions, and a client can add a receiver address by posting a URL to the location

http://.../sensorName/subscription

using the HTTP PUT command. Incoming data is then sent using a POST to the target URL. This interface is slightly different than that described in Section 4, although it achieves the same goals. The interface implementation is currently being changed to match that of Section 4.

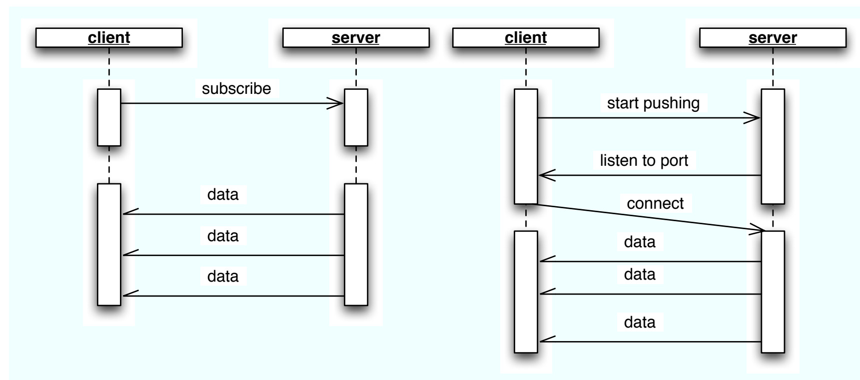

Fig. 4. A publish/subscribe mechanism can be used to efficiently push aperiodic to the client (left) while a new connection can be opened and remain open to push data to a client behind a firewall (right)

The implementation described above is not fundamental to the steam feeds abstraction, and other implementations are possible as long as they provide the same interface and abstraction. For example, a stream feed could be stored in a flat file and served by a normal Web server. Filters could be implemented using syntactic filtering of entries with XML tags that match the filter. Incoming data could be appended to the file by a program that is independent of the Web server software, and the server could server queries over incoming data by observing changes to the file.

One limitation of the publish/subscribe implementation that we describe above is that clients cannot be behind firewalls or network address translation (NAT) devices. An alternative implementation is to negotiate a new long-lived TCP socket between the client and server, and to serve all HTTP responses over that socket. This scheme is compared with the protocol that we implemented for MetroNet in Figure 4. It would be able to stream data to clients behind NATs and firewalls, but has the disadvantage that a socket must be open during the entire query response, which could limit the number of queries that can be supported by the server at a time.

6 Providing Other Interfaces for Stream Feeds

The URL interface is not the only type of interface that can be provided to steam feeds. The URL interface is necessary to make sensor streams first-class citizens on the Web because they can be linked to and they can be crawled and indexed by search engines. However, we can also provide other interfaces by wrapping the URL-based interface. for example, we implemented the interface shown in Figure 1 using SOAP objects, providing a programmatic interface to stream feeds that can be used in any language. In our MetroNet server, the SOAP object for each stream can be located at a URL relative to the location of the stream itself. For example:

```
/streams/frontdoor/temperature/service.wsdl
```

The SOAP interface we provide contains three functions corresponding to the three temporal scopes over which queries can be posed. It also contains a member variable for each TAG in the XML schema used for that stream, so that the programmer can define a filter based on that XML schema. The filter defined through these variables is applied to all queries that are posed with the SOAP object, with any temporal scope. This interface provides the same abstraction as our URL-based interface and therefore provides the same abstraction over a sensor stream, but has the disadvantage that it is not as easily crawlable. Other similar APIs could be defined for stream feeds as necessary by wrapping the URL-based interface.

API	Meaning
get()	query over historical data
getHistorical()	query over incoming data
getIncoming()	query over incoming data
TAG	a member variable through which to apply filters
TAG.eq	return only content where TAG equals a value
TAG.ne	return only content where TAG does not equal a value
TAG.lb	return only content where TAG is greater than a value
TAG.ub	return only content where TAG is less than a value

Table 1. A SOAP interface provides the same functionality as our URL-based interface in a programmatic way. Other interfaces can also be created for stream feeds.

7 Evaluation

We evaluate the efficiency of the stream feed abstraction by measuring the scalability of the stream feed abstraction when applied to a query over a deep fusion tree, and compare our results from using standard Web feeds. We perform this evaluation in two parts. First, we empirically evaluate our server implementation

to establish the trade-off between posing repeated queries for historical data and posing a single query for incoming data. Then, we use these empirical results to drive a simulation in which we pose queries over large fusion trees and measure the latency.

7.1 Repeated Historical Queries vs. Incoming Queries

In our first experiment, we empirically decide between repeatedly querying for the sensor stream and pushing the sensor stream. We setup a Web server that serves a sensor stream that is updated with a frequency of 1Hz. We also setup a set of clients that must receive the data with a frequency from 0.1Hz to 10Hz. The clients filter the query response by indicating the sequence number of the last value received, so only new data is sent in response by the server. We measure the total number of streams that can be served when all clients use push and when all clients use pull. The total values averaged over 100 runs can be seen in Figure 5.

In our experiment, we used the SOAP implementation to measure the performance of the query over incoming data, because the SOAP implementation keeps a socket open for the duration of the push, just as our proposed abstraction would due to problems with firewalls and NATs.

The results show that in the server we implemented, the PUSH queries can support a total of 250 clients at a time, independent of the frequency with which the client needs the data. This is because our web server can only support a maximum of 250 sockets at a time, and the PUSH query keeps the TCP socket open the entire time. This number may differ from servers to servers, due to the capability of the servers.

Theoretically, if each query over historical data only takes 0.2 seconds on the server side, then the server can support a total of more than $5 * 250$ queries in one second because the socket connection of a PULL request lasts less than 0.2 seconds. However, during the experiments we found that the TCP connection number is not the only factor that limits the number of clients we can support per second. As the query frequency increases, with 200 clients, the latency for each query increases due to the speed of database access. For queries over incoming data, a database access only occurs once when any new reading of the stream comes in. Then the server duplicates the new value and pushes it out to all clients. For repeated historical queries, on the other hand, each query requires access to the database because the server doesn't know whether the value of that sensor stream has changed since the previous client's query. This increases the latency of each query and brings down the number of queries we can support per second.

Our results show that more streams can be served using repeated historical queries if the query frequency is less than 1.2Hz. Therefore, since the data was being generated in this experiment at a frequency of 1Hz, we empirically validate that in our system the client should use running queries over incoming data when the sampling frequency is 1.2 times the data generation frequency or more, and should use repeated historical queries otherwise.

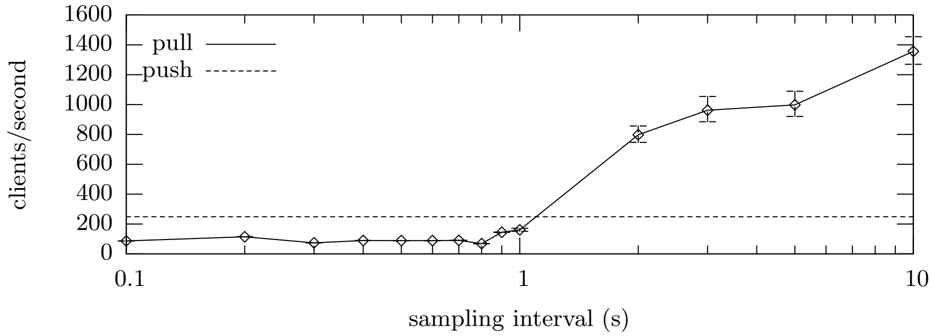

Fig. 5. When clients pull data from the server, the number of clients that can be served increases as the the frequency of requests increases. At a query rate of approximately 1 second, the number of clients served begins to exceed that supported when the server pushes data to the clients in real time.

7.2 Scalability and Efficiency

In our second experiment, we use the results from our first experiment to test the efficiency of stream feeds when used with deep fusion trees in simulation.

With each run, we randomly generate a network of servers with a branching factor of 3 as the depth increases. Although the network resembles a tree, some nodes can have multiple parents assigned to random nodes in the layer above. The nodes are connected in multiple layers from 2 to 15 as shown in Figure 6. The nodes on the bottom are sensors and generate data which is requested by several clients through a series of servers. 5 clients are attached to each server node, and feed the network with queries. The clients are initialized with a random query frequency, and each node has a query rate equal to the sum of the parent's query rates. The latency results reported are the averaged for every client in the network over 5 runs.

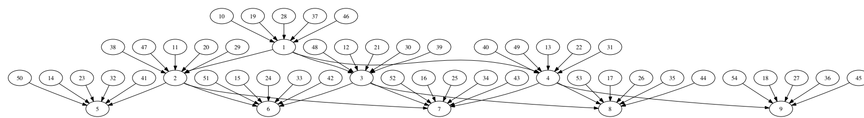

Fig. 6. A randomly generated *fusion tree* of 54 nodes randomly distributed and connected in three layers, with five clients per server.

In our first case, we demonstrate the standard web feed abstraction by restricting servers only to pull for data without server-side filtering from a fusion tree of varying depth. The results from Figure 7 show that the average latency for each query grows exponentially with the depth of the tree. This is a problem,

and demonstrates that normal Web feeds cannot efficiently support deep fusion trees.

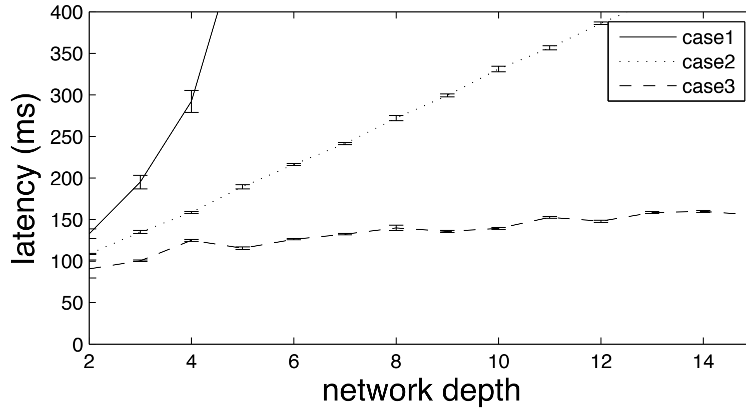

Fig. 7. As the network gets deeper, filtering with the ability to push outperforms the alternatives.

In our second case, we enable servers to use filtering to prevent responses from returning results that are already known by the client. In other words, each client stores the sequence number of the last retrieved data point and passes that as a filter to the server in each request. This experiment shows that the latency grows proportionally with the depth of the network because the filtering significantly reduces the network traffic among all layers during the fusion process, and the latency is dominated by the roundtrip time rather than the transmission time.

In our third case, we evaluate the model with the two streamfeeds extensions, filtering and the ability to push. Each server dynamically decides between requesting historical queries or running queries over incoming data based on its observed query rate and data generation rate. We expect this optimization, by combining the advantages of pushing and pulling, to reduce the average latency of all queries even further than the reduction observed from filters. Because of this dynamic switching there is a balance point at a certain depth in the network, where all the nodes beneath that level begin to push their data to the parents. Because of this region of low latency, the queries suffer most latency when pulling for data in the upper levels of the topology, however more clients can be served in these upper levels. The result is an improvement in the the average latency for all the client's requests. We expect that with larger files, lower sampling rates, and/or deeper fusion trees, this benefit would be further increased.

8 Conclusions

In this paper, we showed that we can provide an efficient abstraction for sensor streams that is a first class citizen on the Web by combining the advantages of Web feed and multimedia streaming paradigms. Each sensor stream has a URL, and parameters encapsulated in this URL allow the user to query for historical data using a PULL model, to query for incoming data using a PUSH model, or both. Other parameters to this URL allow the user to filter the data on the server side, to reduce the size of the data response in repeated queries to the large streaming data objects.

Because this abstraction is a first-class citizen on the web it inherits security, authentication, and privacy mechanisms. Stream feeds can also be crawled and indexed by search engines. The servers are all autonomous, so the server administrators do not need to worry about storing data on the servers of others, or having other data stored on this domain's servers, as with the IrisNet and Hourglass systems. Thus, in this paper, we show that the stream feed abstraction meets the requirements of and can provides the basis for the World Wide Sensor Web, where sensor data can be shared, processed, and published within a single global framework.

References

1. Page, L., Brin, S.: The anatomy of a large-scale hypertextual web search engine. WWW7 / Computer Networks **30** (1998) 107–117
2. Gormon, B., Shankar, M., Smith, C.: Advancing sensor web interoperability. Sensors (April 2005)
3. Geller, T.: Imaging the world: The state of online mapping. IEEE Computer Graphics and Applications **27**(2) (2007) 8–13
4. Aberer, K., Hauswirth, M., Salehi, A.: A middleware for fast and flexible sensor network deployment. In: VLDB '06: Proceedings of the 32nd international conference on Very large data bases, VLDB Endowment (2006) 1199–1202
5. Rose, I., Murty, R., Pietzuch, P., Ledlie, J., Roussopolous, M., Welsh, M.: Cobra: Content based filtering and aggregation of blogs and rss feeds. In: 4th USENIX Symposium on Networked Systems Design and Implementation. (2007)
6. Schulzrinne, H., Rao, A., Lanphier, R.: Real time streaming protocol (rtsp). Technical report, Network Working Group (1998)
7. P. Saint-Andre, E.: Extensible messaging and presence protocol. Technical report, Jabber Software Foundation (October 2004)
8. Hickson, I.: Html 5. Technical report, Google (September 2007)
9. Armenio, F., Barthel, H., Burnstein, L., Dietrich, P., Duker, J., Garrett, J., Hogan, B., Ryaboy, O., Sarma, S., Schmidt, J., Suen, K., Traub, K., Williams, J.: The epcglobal architecture framework. Technical report, EPCglobal (2007)
10. Cooper, O., Edakkunni, A., Franklin, M.J., Hong, W., Jeefrey, S., Krishnamurthy, S., Reiss, F., Rizvi, S., Wu, E.: Hifi: A unified architecture for high fan-in systems. Proceedings of the 30th VLDB Conference (2004)
11. Gibbons, P.B., Karp, B., Ke, Y., Nath, S., Seshan, S.: Irisnet: An architecture for a world-wide sensor web. IEEE Pervasive Computing **2**(4) (2003) 22–33

12. Huebsch, R., Hellerstein, J.M., Boon, N.L., Loo, T., Shenker, S., Stoica, I.: Querying the internet with pier (2003)
13. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., Zdonik, S.: Scalable Distributed Stream Processing. In: CIDR 2003 - First Biennial Conference on Innovative Data Systems Research, Asilomar, CA (January 2003)
14. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.: Telegraphcq: Continuous dataflow processing for an uncertain world (2003)
15. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: a scalable continuous query system for Internet databases, ACM SIGMOD (2000) 379–390
16. Liu, S.N.J., Zhao, F.: Challenges in building a portal for sensors world-wide, First Workshop on World-Sensor-Web: Mobile Device Centric Sensory Networks and Applications (WSW'2006) (2006)
17. Shneiderman, J., Pietzuch, P., Ledlie, J., Roussopolous, M., Seltzer, M., Welsh, M.: Hourglass: An infrastructure for connecting sensor networks and applications. Technical report, Harvard University (2004)