# A Sensor Observation Service Extension for Internet of Things

Argyrios Samourkasidis and Ioannis N. Athanasiadis(✉)

Information Technology Group, Wageningen University,
Hollandseweg 1, 6706 KN Wageningen, The Netherlands
{argyrios.samourkasidis,ioannis.athanasiadis}@wur.nl

**Abstract.** This work contributes towards extending OGC Sensor Observation Service to become ready for Internet of Things, i.e. can be employed by devices with limited capabilities or opportunistic internet connection. We present an extension based on progressive data transmission, which by-design facilitates selective data harvesting and disruption-tolerant communication. The extension economizes resources, while respects the SOS specification requirement that the client should have no a-priori knowledge of the server capabilities. Empirical experiments in two case studies demonstrate that the extension adds little overhead and may lead to significant performance improvements in certain cases, as for irregular timeseries. Also, the proposed extension is not invasive and backwards compatible with legacy clients.

**Keywords:** Open Geospatial Consortium · Sensor Observation Service · Internet of Things · Syntactic interoperability · SOS 2.0 · Sensor Web · Progressive transmission · Pagination · Timeseries data

## 1 Introduction

Internet of the Things (IoT) is a dynamic, open, participatory ecosystem of decentralized and collaborative devices. Recent technological advances resulted in a plethora of low-cost devices with extended capabilities compared to traditional sensors. New generation of devices are miniaturized and empowered with storage, processing and networking capacity. They are essentially transformed into **smart nodes**, that operate autonomously, may offer added value services [31], and collaborate with each other in the cloud [4]. Smart nodes could offer capture, storage and dissemination services of sensory information in a single device [36]. IoT devices are also instrumental to the proliferation of new data sources [14], sharing of information [15], and contribute to the *big data* movement. Internet of Things advances the vision of Sensor Web, *an infrastructure which enables interoperable usage of sensor resources* [8]. In the IoT era, Sensor Web is challenged to offer services that are interoperable, but at the same time perform efficiently with **less resources**, saving processing power and network bandwidth.

I. Podnar Žarko et al. (Eds.): InterOSS-IoT 2016, LNCS 10218, pp. 56–71, 2017.
DOI: 10.1007/978-3-319-56877-5_4

Interoperable data interchange for sensor data has been driven by the Open Geospatial Consortium (OGC). OGC introduced service interfaces and information models within Sensor Web Enablement (SWE), which is founded on machine-to-machine communication [5,7]. Service interfaces, as the Sensor Observation Service, Web Feature Service, Web Coverage Service, SensorThings provide interoperable means for geospatial information discovery and retrieval. Sensor Observation Service (SOS) [24,25] is an OGC service interface, which promotes interoperable sensor-borne data exchange, operates as a web service, and supports for syntactic and semantic interoperability.

In the IoT era, architectural paradigms and technologies need to respect the limited capabilities of devices. The SWE 2.0 has been established with technologies as the Simple Object Access Protocol (SOAP) and XML-based information models, which are considered to add substantial overhead - a critical issue for IoT devices. On the other hand, Representational State Transfer (REST) and JSON-based information models seem to provide services which excel over SOAP and XML, in terms of power consumption and performance [22]. Beyond these technical limitations, there are certain *design* choices that preclude SOS as an appropriate IoT outlet.

In this paper, we investigate current SOS design and propose an extension. In Sect. 2, we present related work, how SOS operates and challenges identified in the literature. In Sect. 3, we identify SOS design shortcomings from an IoT perspective, and introduce a pagination technique in order to promote selective data harvesting, enable seamless data integration and facilitate machine-to-machine interoperability. Section 4 presents an implementation and details the two case studies, which were designed to test the efficiency of the extension, along with experimental results. Section 5 provides with a discussion about our findings and contributions, concludes the research and lays the groundwork for future work.

## 2 Related Work

### 2.1 Service Orientation and Interoperability in Sensor networks

Service-Oriented Architecture (SOA) is an architectural paradigm founded on self-describing, self-contained services. Key concept in SOA is that services may be developed, maintained and served by different entities, and can subsequently be combined and produce composite applications. SOA has been instrumental for highly interoperable systems, as services are platform and language independent [30].

In the frame of interoperable data interchange, OGC introduced Sensor Web Enablement (SWE), which follows the SOA architectural paradigm. Standards developed within SWE provide means for the discovery and retrieval of sensor observations. SWE contributes towards the vision of Sensor Web, where web-accessible sensor networks and archived sensor observations can be discovered and accessed using standard protocols and application program interfaces (APIs) [5]. They are realized through *web services*, i.e. services "identified by a URI, whose service description and transport utilize open Internet standards" [30]. Communication between service interfaces and other services or clients is achieved through Simple Object Access Protocol (SOAP), which builds on existing communication layers (i.e. HTTP) [10]. SWE is a very important infrastructure [8] as it offers interoperable protocols for advertising, disseminating and requesting data among heterogeneous sensor systems and devices.

## 2.2 The Sensor Observation Service

Sensor Observation Service (SOS) is an OGC service interface specification for accessing sensor observations, which acts as "the intermediary between a client and an observation repository" [5]. SOS interface enables clients to request, filter and retrieve observations, and metadata about repositories and sensors.

SOS comes with a *core* set of services, and *extensions* that enrich it with extra functionality, or *profiles* for domain-specific behavior. The current 2.0 specification [24] defines three *core* operations:

a. service discovery (`GetCapabilities`),
b. sensors metadata retrieval (`DescribeSensor`), and
c. observations retrieval (`GetObservation`).

There are several extensions and profiles available, but their description falls outside the scope of this paper. As an indicative example for the reader, the *transactional extension* provides with services to register new sensors and add new observations.

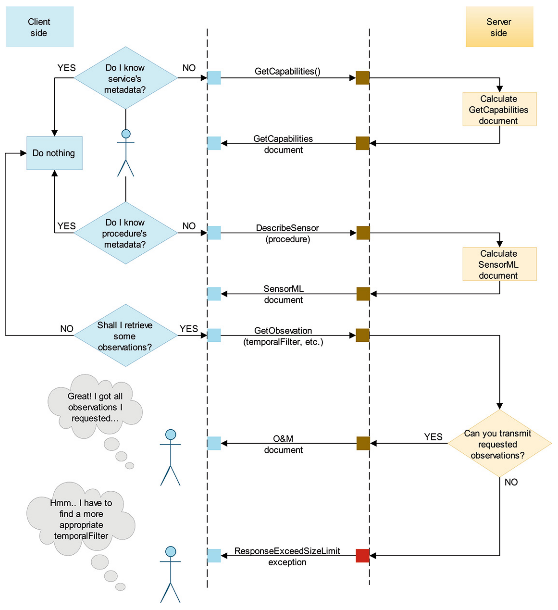
Client side
Server side
YES
Do I know service's metadata?
NO
GetCapabilities()
Calculate GetCapabilities document
Do nothing
GetCapabilities document
YES
Do I know procedure's metadata?
NO
DescribeSensor (procedure)
Calculate SensorML document
SensorML document
NO
Shall I retrieve some observations?
YES
GetObsevation (temporalFilter, etc.)
Great! I got all observations I requested...
O&M document
YES
Can you transmit requested observations?
NO
Hmm.. I have to find a more appropriate temporalFilter
ResponseExceedSizeLimit exception

**Fig. 1.** A typical observation retrieval workflow using SOS

SOS is a pull-based service interface and is intended for machine-to-machine communication. The protocol prescribes a communication between a client and a server, both can be considered to be software agents. The client submits a request and the server answers with a response, typically in the form of XML document. Responses are encoded in appropriate SWE related XML schemas as Observation & Measurements [23], or SensorML [27]. A typical observation retrieval workflow using SOS is depicted in Fig. 1. First, the client inquires the server for its capabilities. Then, it may ask for descriptions on certain sensors, and finally requests for observations from one or more sensors. A typical `GetObservation` request includes temporal and/or spatial boundaries.

When SOS server encounters an error while performing a `GetObservation` operation, it returns an exception. For example, if client asks for wrong values of arguments an `InvalidParameterValue` exception is rendered. In the current SOS 2.0 interface standard [24] there is also a type of exception for the cases that the response exceeds a **size limit**. We will investigate this further below.

## 2.3 Challenges in Sharing Sensor Observations in IoT

Internet of Things consists of *smart nodes* equipped with sensors and network connectivity, able to interact with their environment and share information. *Smart nodes* are entitled with specific characteristics:

a. restrained capabilities (in terms of energy and processing power),
b. opportunistic Internet connection, and
c. heterogeneity in resulting data formats and communication protocols [3].

Key challenges towards the IoT realization include energy efficiency, integration of service technologies and security/privacy [21]. Also, thematic and spatial concerns of deployed IoT systems pose great challenges in spatiotemporal aggregation of disperse observation datasets.

As regards with **heterogeneous sensor integration**, previous studies have been conducted towards various directions. A virtual integration framework for heterogeneous meteorological and oceanographic data sources is demonstrated in [33]. A *SOS profile* to facilitate multi-agency sensor data integration was reported in [1,19]. Fredericks et al. argue in [12] that quality metadata should also be transmitted through SWE services, in order the realization of automatic data integration to be achieved.

Integration of spatially diverse sensor timeseries utilizing OGC standards concerned Horita et al. in [16]. They developed a spatial decision support system for flood risk management, associating Volunteered Geographic Information (VGI) and measured data derived from Wireless Sensor Networks (WSNs). Data acquisition, integration and dissemination is orchestrated by a SOS instance.

Only recently, OGC introduced *SensorThings API* to facilitate "the interconnection of IoT devices, data, and applications over the Web" [28]. In contrast with other OGC standards, SensorThings API adopts the REST paradigm and utilizes JSON-based information models. SensorThings API defines HTTP requests to facilitate observations' retrieval, as well tasking of sensors and actuators.

Using parameters to regulate response size to requests within OGC-related standards, was a topic of interest for [26,28,29]. Lengthy responses to `GetObservation` requests have been identified as a potential danger to both SOS server and clients [26]. In the same work, it has been indicated that beyond the `ResponseExceedSizeLimit` exception, other certain limitations as regards with the number of returned observation should be concerned and imposed. The WFS interface standard [29] and the SensorThings API offer a paging implementation, that allows the client to limit the number of features included in a response by using two optional arguments (*count*, *startindex* for WFS, and *top*, *skip* for SensorThings API).

Last but not least, several researchers investigated the suitability of limited bandwidth, energy, and processing power devices to host a SOS server. These have mainly concentrated on (a) adoption of lightweight architectural paradigms (e.g. REST instead of SOAP [17,35,39]), and (b) evaluation of SOS lightweight implementations [18,32]. We have also deployed SOS over a Raspberry Pi to exploit the potential of low-cost embedded devices [36].

In this work we concentrate on the SOS service interface design and evaluate the efficiency of communication between client and server.

## 3 Methods

### 3.1 SOS Service Interface Design Issues

According to SOS specification, clients are not allowed to know sensor observations' frequency. The server advertises the boundaries of the information it holds, but not the resolution. Any client is not possible to infer the sensor temporal or spatial resolution, based on their communication with the server. This requirement is that the client has access with **no a-priori knowledge** [25]. While this enforces reusability and generality of the service interface, it may lead to excessive data requests, which may result to server overload, or even Denial of Service attacks.

Excessive data transmission has been identified as an issue for `GetObservation` requests. In the first specification of SOS, there was not imposed any limitation, regarding the maximum number of observations which could be transmitted. For the server, the only viable response to of a `GetObservation` request was to return a set of observations. The server had no way to refuse to respond, in cases where the client was asking for an excessive amount of data, it was busy, or any other reason.

To illustrate the above shortfall we will consider a service offered by National Oceanic and Atmospheric Administration (NOAA) [9]. NOAA's Center for Operational Oceanographic Products and Services (CO-OPS) offers openly a variety of sensor observations using SOS. In this implementation, if a client requests observations for a time range which exceeds 31 days, the server responds with an exception, rejecting the parameter value:

```
<Exception exceptionCode="InvalidParameterValue"
    locator="eventTime">
        <ExceptionText>
            Max 31 days of data can be requested.
            62.0 days were requested.
        </ExceptionText>
</Exception>
```

Note that the exception rejects the parameter value, disclosing in a non machine interoperable message of the size limits for this request.

In the future work section of SOS 1.0 specification [25] it was acknowledged that: *"The density of requests and offerings must be addressed,... so that large data volumes are not transmitted unnecessarily due to a lack of information about service offerings."*. Indeed, that was addressed in SOS 2.0 by introducing an *exception* to manage excessive data requests, while taking into consideration the *no a-priori knowledge* requirement [5]. The `ResponseExceedSizeLimit` exception functionality resembles the response of NOAA server above, but with pertinent semantics to the exception thrown: The server is able to inform the client that the *"requested result set exceeds the response size limit of the service and thus cannot be delivered"* [24]. Both server and client applications are protected from extremely big response sizes, and the *no a-priori knowledge* requirement is respected.

The `ResponseExceedSizeLimit` exception of SOS 2.0 is a significant improvement compared to SOS 1.0, as it allows the server to respond to a request with an exception than with actual data. Note that, the response size limit should not be considered a fixed parameter. It could change when there is high traffic, or service maintenance. In those conditions, the server should be allowed to not to respond to requests that would under normal conditions.

However, the main limiting factor to this design is that clients have no insights regarding the carrying capacity of the server, or (equivalently) the density of an offering. Due to the *no a-priori knowledge* requirement, clients cannot infer how to narrow down their requests so that server responds.

We identify two cases here. First case is when the server publishes regular sensor observations. Under this category fall most long-term, permanent sensor infrastructures. In this case, clients could implement heuristic techniques to discover the response size limit (assuming that it is constant).

In the second case, observation streams are irregular. This may happen if the sensor sampling frequency varies, or sensors move. For example, consider sensors operating in energy restrained environments and adopt opportunistic sensing techniques, or event-based sensing [2]. Volunteered Geographic Information Systems which enable individuals [11,13] or cars [6] as data providers, fall in the same case. In these situations, it is impossible for the client to make any kind of estimate on the response size, and devise a strategy to reduce accordingly the spatiotemporal boundaries of their query.

Responding with an exception to voluminous requests could be tolerated in fixed sensor networks (case one above). However, it hinders SOS applicability in resource-constrained environments. As clients are neither aware of the response size limits, nor how to restrict their queries, the SOS communication protocol underperforms: It wastes both processing power and network bandwidth as it is engaged in more request/response cycles. This, ultimately results in bigger response times. Such drawbacks are incompatible with the Internet of Things needs. This problem could be addressed by introducing a progressive data transmission technique described below.

## 3.2 The Resumption Token Technique and Open Archives Initiative

The notion of selective data retrieval was introduced in Lagoze and Van de Sompel [20]. Utilizing a *resumption token*, large and resource-demanding data transactions are fragmented into several requests/responses. The client submits a request and the server responds with a part of the result and a *resumption token*. Then the client (harvester) can use this *resumption token* in follow up requests to get the following part of its initial request. Gradually, by consecutive requests the client retrieves the all the partial answers to its initial request. This mechanism enables the server to handle with requests that have large responses, with respect to available bandwidth and/or processing power.

## 3.3 A Pagination Extension for SOS

SOS service interface can address IoT needs by introducing progressive data transmission. We extend the current SOS service interface with a *resumption token* parameter in the `GetObservation` requests. By fragmenting requests into many sequential ones, we transform SOS into a **disruption-tolerant** service interface, as clients are enabled to ask for specific observation subsets. Observations are divided and loosely packed into *pages* of certain size. The number of observations contained in a *page* (i.e. chunk of subsequent observations) is determined by the SOS server.

The observation retrieval workflow according to the proposed design is depicted in Fig. 2. The client asks for a set of observations with a `GetObservation` request. The server processes the request, and always responds with an O&M document. If the response exceeds the carrying capacity of the server, results will be organized in subsets (called pages), and the server response will include an additional element, called `next` which will point to the URL of the next page of results. The next page URL is the same as the original request, but contains an extra parameter called page, which has the role of the resumption token. The `page` parameter is optional: when a client request does not contain a `page` argument, the server responds with the first *page* of the request. The last page of the parts contains no next page element to notify the client of the end of the transmission.

In the simplest case, server carrying capacity could be an arbitrary, fixed threshold, similar to the *request size limit* of the SOS 2.0 exception. Of course, the server carrying capacity may dynamically vary according to result set properties, or server resources, enabling network load balancing, efficient use of energy, etc.

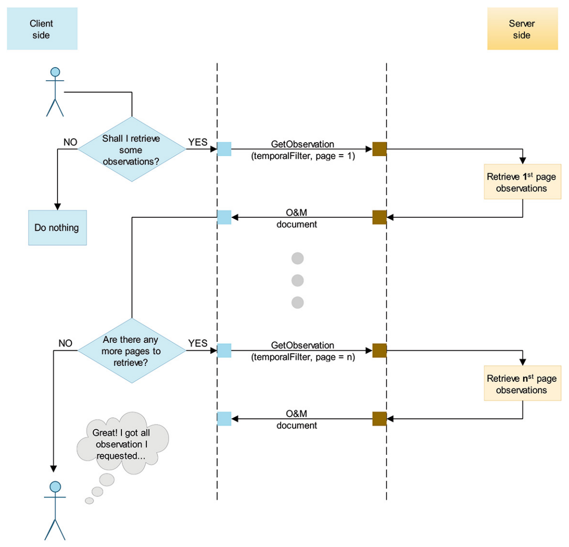
Client side
Server side
Shall I retrieve some observations?
NO
YES
Do nothing
GetObservation (temporalFilter, page = 1)
Retrieve 1st page observations
O&M document
Are there any more pages to retrieve?
NO
YES
GetObservation (temporalFilter, page = n)
Retrieve nst page observations
O&M document
Great! I got all observation I requested...

**Fig. 2.** A typical *paginated* observation retrieval workflow

It could even change during the transmission, as the total number of pages is not disclosed to the client. The `page` resumption token could be constructed incrementally as page number in case the server has a fixed carrying capacity, and data do not change. In case of varying page size, the `page` parameter can take unique pseudo-random integer values. In case where data changed during the communication, or any other reason, the next `page` token could be revoked by the service provider.

## 3.4 Expected (by Design) Benefits

The paginated protocol proposed here is beneficial for both server and client **efficiency** and **performance**. The communication protocol does not waste resources to respond with exceptions, as all requests result to responses that carry observations. This saves processing power and communication bandwidth in both client and the server.

Another attribute of the design we propose is its **non-invasive** nature. Given the *page* parameter is optional, current SOS clients can seamlessly submit `GetObservation` requests and retrieve observations, as long as the SOS server carrying capacity is not exceeded. This means that existing SOS 1.0 or 2.0 server infrastructures could switch to a paginated implementation, and as long as they do not change their size limit threshold, existing clients would continue to operate without disruption. In the rest cases, a *page-parser* method should be implemented and incorporated in legacy clients. This method would parse a `GetObservation` response document to determine the URL of the next `GetObservation` request. On server side, the *pagination* extension could be easily applied on top of existing implementations.

## 4 Demonstration and Implementation

### 4.1 Setup

The SOS pagination extension introduced above comes with design advantages discussed in the previous section. There are also performance improvements that we experimentally evaluated by setting up two case studies. Without loss of generality, we assume not movable sensors that hold timeseries information. In case study one, the server holds a regular timeseries dataset, while in the second case study an irregular one. For both cases, we compared the SOS pagination extension (**SOS-p**) service interface against *SOS 2.0*.

The *SOS-p* server is queried by a corresponding client (*PAC*), that is able to handle `page` resumption tokens. For *SOS 2.0* server, we considered two clients: one that is not aware of *SOS 2.0* carrying capacity and finds it by employing a divide-and-conquer algorithm (*DAC*); and one that has this a-priori knowledge (*LEC*).

The three clients are in detail as follows:

**Divide and Conquer client (DAC):** *DAC* submits `GetObservation` requests according to *SOS 2.0* specification. When the server responds with a `Response-ExceedSizeLimit`exception, *DAC* halves the time window and submits a new query. When *DAC* finds a time window for which the server responds with no exception, it continues asking for observations with of this duration size in the temporal filter, until it has received all the data corresponding to the original request.

**Leaky client (LEC):** *LEC* knows the server *carrying capacity* and arranges the *temporal filter* of its request, so that there are no exceptions. While this is against the *no a-priori knowledge* requirement, it corresponds to the most favorable situation for the existing *SOS 2.0* protocol. *LEC* submits `GetObser-vation` requests to *SOS 2.0* only for case study 1.

**Pagination-aware client (PAC):** *PAC* client submits `GetObservation` requests according to *SOS-p*, i.e. it is capable of processing the page resumption token. In its first `GetObservation` request asks for the first page, and then processes the response for the next `page` it will ask for. If the `GetObservation` response document does not contain a next *page* tag, it means that all requested observations were transmitted.

### 4.2 Implementation and Synthetic Datasets

This study makes use of the AiRCHIVE SOS server implemented in Python [36]. Clients were also implemented in Python. Queries to SOS server were submitted as *HTTP GET* requests via Python Requests module [34]. Response times for each case study were facilitated using the Python Time module [38]. All experiments were carried out on a Intel Core i5 4 Mac with a 2,4 GHz and 16.0 GB of memory (1600 MHz DDR3), running OS X El Capitan (Version 10.11.1). SOS server and SOS client instances operated on the same physical machine.

In both case studies, a dataset of 15,000 observations was artificially generated. In case study one, we assumed that measurements are sensed in constant intervals of 10 s. In case study two, observations were timed with a *inconstant* frequency. Observation time interval varies from 10 to 3000 s, distributed uniformly. Timestamps were generated with the Python Random Number generator module, using Mersenne Twister [37]. Both timeseries were stored in two SQLite databases and made available to the servers.

### 4.3 Experimental Setup and Metrics

As limited bandwidth and processing power are key elements of IoT systems, we set up accordingly our experiments. The **carrying capacity** of the servers was defined to be 15 observations. This arbitrary threshold was chosen so that there will be significant traffic of SOS requests. *SOS 2.0* server would render a `ResponseExceedSizeLimit` exception if the result set would include more than 15 observations. *SOS-p* server organizes its responses in pages of 15 observations per page.

Clients were configured to request for observations for time intervals that result to 1 000, 2 000, 4 000, 8 000 or 15 000 observations (*response length*). Experiments have been repeated 10 times for all clients and both case studies.

For all experiments, we recorded two metrics:

a. the **response time** is the total time passed until the client has received the total amount of data requested. Measured in *seconds*.
b. **transfer volume** is the total size of all response documents received by the client until the whole response has been received. It is measured in *MB*.

Response times are averaged across the 10 repetitions, while transfer volume is the same for each repetition.

For the cases of **SOS 2.0** implementation, in the *average response time* and *transfer volume*, time spent and resulted size of exceptions are also included.

### 4.4 Experimental Results

Tables 1 and 2 summarize the results for both case studies and all clients. The *response time* is reported as average and standard deviation of ten repetitions.

For case study 1, best results are achieved, as expected, by the client that is aware of the server carrying capacity (*LEC*), but violates the *no a-priori knowledge* requirement. The divide-and-conquer client (*DAC*) in *SOS 2.0* adds an overhead to the transmission, as it needs to search for a working time interval. Its performance is affected mostly of how close the time interval found is to the servers carrying capacity. The response time was significantly increased in

**Table 1.** Experimental results for the **regular** timeseries for all three clients. Average response times and standard deviation across ten requests are reported. Total volume of the data transmitted, number of requests, and number of exceptions for *DAC*.

| Query length | PAC | | | LEC | | DAC | | |
|---|---|---|---|---|---|---|---|---|
| | Resp. time (std) [s] | Vol [MB] | Reqs | Resp. time (std) [s] | Vol [MB] | Resp. time (std) [s] | Vol [MB] | Exceptions |
| 1000 | 1.34 (±0.049) | 0.59 | 67 | 1.29 (±0.013) | 0.58 | 2.36 (±0.02) | 0.63 | 7 |
| 2000 | 2.68 (±0.012) | 1.2 | 134 | 2.57 (±0.011) | 1.2 | 4.64 (±0.05) | 1.3 | 8 |
| 4000 | 5.53 (±0.083) | 2.4 | 267 | 5.22 (±0.017) | 2.3 | 9.27 (±0.03) | 2.5 | 9 |
| 8000 | 11.93 (±0.031) | 4.7 | 534 | 10.31 (±0.034) | 4.7 | 18.31 (±0.06) | 5.0 | 10 |
| 15000 | 24.77 (±0.739) | 8.9 | 1000 | 19.05 (±0.043) | 8.7 | 21.33 (±0.06) | 8.8 | 10 |

our experiments in Table 1. In the contrary, the performance of *SOS-p* and the paginated client *PAC* is very close to the server carrying capacity, without any breach of the *no a-priori knowledge* requirement. Experimental results in Table 1 illustrate overheads less than 5% in response time for up to few hundreds of pages, while for bigger numbers of requests overheads in time may end up to 30% in response time. This is attributed to the efficiency of the pagination implementation and is a well-known limitation among the database community. In the future, we will investigate other database options that can improve this further.

For case study 2, irregular timeseries are served therefor there is no notion of leaking the prior knowledge of the server carrying capacity. Here the paginated *SOS-p* excels over *SOS 2.0*, as presented in Table 2. *SOS-p* and *PAC* are faster than *SOS 2.0* by more than 60% on average on every `GetObservation` request. Also, note that number of requests has been roughly doubled, which results to a noticeable difference in the amount data transmitted. This is to be expected, as the *divide and conquer* strategy may end up finding a query window that is far from what can be actually served. There could be other search algorithms employed for improving *DAC* performance. However, it is made clear from this

**Table 2.** Experimental results for the **irregular** timeseries. For *PAC* and *DAC* clients reports average response times and standard deviation across ten requests. Total volume of the data transmitted, number of requests and number of exceptions for *DAC*.

| Query length | PAC | | | DAC | | |
|---|---|---|---|---|---|---|
| | Resp. Time (std) [s] | Vol [MB] | Reqs | Resp. Time (std) [s] | Vol [MB] | Exceptions |
| 1000 | 1.35 (±0.02) | 0.59 | 67 | 2.40 (±0.03) | 0.63 | 7 |
| 2000 | 2.75 (±0.05) | 1.2 | 134 | 4.71 (±0.05) | 1.3 | 8 |
| 4000 | 5.66 (±0.07) | 2.4 | 267 | 9.28 (±0.06) | 2.5 | 9 |
| 8000 | 11.97 (±0.08) | 4.7 | 534 | 18.34 (±0.03) | 5.0 | 10 |
| 15000 | 24.62 (±0.11) | 8.9 | 1000 | 36.81 (±0.88) | 9.5 | 11 |

experiment, that the paginated protocol guarantees **by design** that the optimal number of measurements is included in each response. *SOS-p* entrusts the burden of coordinating the observation boundaries to the server, which knows its limits, than having the client wasting resources with requests of suboptimal lengths. The improved performance ensures that there is no waste of resources on both the client and the server side.

## 5 Discussion and Conclusions

This work contributes towards improving OGC SOS protocol to become IoT ready. Drafting on top of IoT requirements as efficient resource utilization and opportunistic Internet connection, and taking into consideration response size to `GetObservation` requests requirements set in [26], we designed a SOS extension, which implements a *pagination* mechanism.

There is a fundamental difference between our design and the paging mechanism introduced in OGC WFS [29]. WFS paging design contradicts with the rationale of SOS `ResponseExceedSizeLimit` exception, that is to enable the SOS server to manage efficiently its resources. Conversely, it allows clients to select the number of returned observations, which is a feature that can only facilitate specific applications (e.g. Graphical User Interfaces which can visualize a certain number of observations). In the contrary, the solution proposed in this work follows the Open Archives Initiative design pattern, and the decision on the *page size* remains with the server, not the client. As we demonstrated above, this is a necessary condition for the server in the IoT era, as it allows for parsimonious use of resources, and protection from queries resulting with very big results.

*Pagination* introduces the notion of **progressive transmission**, which fits for purpose with timeseries data sequential nature, but is also suitable for any kind of spatiotemporal requests. It adds **disruption-tolerance** as an additional SOS feature, since a client can request for and retrieve a specific page. This is very useful when big datasets are to be transmitted or when the Internet connection is poor. Our design enables a SOS server to exploit its resources to the maximum, as computational power and network bandwidth are spent for yielding results, not for handling exceptions. Thus, the paginated extension enables **by-design** SOS for devices with restrained capabilities, where resources are economized in sharing interoperable knowledge.

Whilst our suggested design entails new improvements to the existing *SOS 2.0*, its importance is highlighted by its **non-invasive** nature. Backwards compatible design is achieved through the *optional* page parameter, since all requested data could be included in one *page*. This way, current *SOS 2.0* clients could operate without further modifications with *SOS-p* extended servers, if the server always responds with the whole data requested.

Evaluating the *SOS-p* extension against specific metrics, we validated improvements by experiments. Those improvements are mainly concerned with efficiency. Lower `GetObservation` requests completion times contribute towards

IoT devices energy conservation, since computational resources are occupied for less time, and thus more clients can be served simultaneously. In addition to that, when carrying capacity is not known to the client, the *SOS 2.0* protocol is under-operating, as possibly transmits less observations in each request. This results to more request-response transactions, with overheads in data volume and duration time.

The pagination extension introduced here offers a remedy to *SOS 2.0* shortfalls in handling exceptions, by providing a machine interoperable solution. It also fills-in the SOS missing piece, that is to "*allow a client to determine the density of an offering*" [25]. Instead of that, it delegates to the server to drive protocol.

Advancements discussed so far lay the groundwork for future work. Firstly, our intention to use *pagination* was exploratory, thus there is room for further improvements in the implementation to further improve performance. One direction for improvement is the adoption of a caching mechanism. *Pagination* is a good candidate for caching techniques, since requests are incremental and queries are submitted sequentially. With the design introduced here, the client reveals its intentions to the server, by asking the whole spatiotemporal boundaries of interest. If the response is too big, the server will return the first page that includes a part of the results. As the client intentions have been disclosed to the server, this allows for caching mechanisms to be set up on the server side.

Following the anonymous reviewer comments and the discussions during the InterOSS-IoT Workshop in Stuttgart on November 7th, 2016, authors will consider bringing this forward to OGC for consideration as a *white paper*.

To summarize, we argued that current SOS design was not intended for the Internet of the Things era. We designed a pagination extension offering progressive data transmission, economizing resources and tackling with limited or interrupted Internet connectivity with a disruption-tolerant protocol, while respecting SOS specification. There is a small effort into extending current SOS servers and clients to implement the pagination extension, while there are significant performance improvements, as indicated by the experimental results. The pagination extension sets the grounds for enabling SOS as an Internet of the Things dissemination outlet for sensor observations.

## Supplementary Materials

Pagination cumulative results are available on Zenodo:
http://doi.org/10.5281/zenodo.178913

**Acknowledgements.** This work was partially supported by Greek General Secretariat for Research and Technology, grant 11SYN-6-411 (ALPINE), and the European Community's Seventh Framework Programme grant 613817 (MODEXTREME). Authors are grateful to the four anonymous reviewers for their valuable feedback and the participants of the InterOSS-IoT Workshop (Stuttgart, Nov. 7th, 2016) for their comments.

## References

1. Alamdar, F., Kalantari, M., Rajabifard, A.: Towards multi-agency sensor information integration for disaster management. Comput. Environ. Urban Syst. **56**, 68–85 (2016). http://dx.doi.org/10.1016/j.compenvurbsys.2015.11.005
2. de Assis, L.F.F.G., Behnck, L.P., Doering, D., de Freitas, E.P., Pereira, C.E., Horita, F.E.A., Ueyama, J., de Albuquerque, J.P.: Dynamic sensor management: extending sensor web for near real-time mobile sensor integration in dynamic scenarios. In: Proceedings of International IEEE Advanced Information Networking and Applications (AINA), pp. 303–310, March 2016. http://dx.doi.org/10.1109/AINA.2016.100
3. Atzori, L., Iera, A., Morabito, G.: The Internet of Things: a survey. Comput. Netw. **54**(15), 2787–2805 (2010). http://dx.doi.org/10.1016/j.comnet.2010.05.010
4. Botta, A., de Donato, W., Persico, V., Pescapé, A.: Integration of cloud computing and Internet of Things: a survey. Future Gener. Comput. Syst. **56**, 684–700 (2016). http://dx.doi.org/10.1016/j.future.2015.09.021
5. Botts, M., Percivall, G., Reed, C., Davidson, J.: OGC® sensor web enablement: overview and high level architecture. In: Nittel, S., Labrinidis, A., Stefanidis, A. (eds.) GSN 2006. LNCS, vol. 4540, pp. 175–190. Springer, Heidelberg (2008). doi:10.1007/978-3-540-79996-2_10
6. Broering, A., Remke, A., Stasch, C., Autermann, C., Rieke, M., Möllers, J.: enviroCar: a citizen science platform for analyzing and mapping crowd-sourced car sensor data. Trans. GIS **19**(3), 362–376 (2015). http://dx.doi.org/10.1111/tgis.12155
7. Bröring, A., Echterhoff, J., Jirka, S., Simonis, I., Everding, T., Stasch, C., Liang, S., Lemmens, R.: New generation sensor web enablement. Sensors **11**(3), 2652–2699 (2011). http://dx.doi.org/10.3390/s110302652
8. Bröring, A., Janowicz, K., Stasch, C., Schade, S., Everding, T., Llaves, A.: Demonstration: a RESTful SOS proxy for linked sensor data. In: Proceedings of 4th International Workshop on Semantic Sensor Networks (SSN11), pp. 123–126 (2011)
9. Center for operational oceanographic products and services (co-ops) sensor observation service (2017). https://opendap.co-ops.nos.noaa.gov/ioos-dif-sos/. Accessed 22 Jan 2017
10. Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S.: Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI. IEEE Internet Comput. **6**(2), 86 (2002). http://dx.doi.org/10.1109/4236.991449
11. Drosatos, G., Efraimidis, P., Athanasiadis, I., Stevens, M., D'Hondt, E.: Privacy-preserving computation of participatory noise maps in the cloud. J. Syst. Softw. **92**, 170–183 (2014). http://dx.doi.org/10.1016/j.jss.2014.01.035
12. Fredericks, J.J., Botts, M., Cook, T., Bosch, J.: Integrating standards in data QA/QC into OpenGeospatial consortium sensor observation services. In: Proceedings of OCEANS 2009-EUROPE, pp. 1–6, May 2009. http://dx.doi.org/10.1109/OCEANSE.2009.5278211
13. Goodchild, M.F.: Citizens as sensors: the world of volunteered geography. GeoJournal **69**(4), 211–221 (2007). http://dx.doi.org/10.1007/s10708-007-9111-y
14. Hashem, I.A.T., Yaqoob, I., Anuar, N.B., Mokhtar, S., Gani, A., Khan, S.U.: The rise of "big data" on cloud computing: review and open research issues. Inf. Syst. **47**, 98–115 (2015). http://dx.doi.org/10.1016/j.is.2014.07.006
15. Havlik, D., Bleier, T., Schimak, G.: Sharing sensor data with SensorSA and cascading sensor observation service. Sensors **9**(7), 5493–5502 (2009). https://dx.doi.org/10.3390/s90705493

16. Horita, F.E., de Albuquerque, J.P., Degrossi, L.C., Mendiondo, E.M., Ueyama, J.: Development of a spatial decision support system for flood risk management in Brazil that combines volunteered geographic information with wireless sensor networks. Comput. Geosci. **80**, 84–94 (2015). https://doi.org/10.1016/j.cageo.2015.04.001
17. Janowicz, K., Bröring, A., Stasch, C., Schade, S., Everding, T., Llaves, A.: A RESTful proxy and data model for linked sensor data. Int. J. Digit. Earth **6**(3), 233–254 (2013). http://dx.doi.org/10.1080/17538947.2011.614698
18. Jazayeri, M.A., Liang, S.H., Huang, C.Y.: Implementation and evaluation of four interoperable open standards for the Internet of Things. Sensors **15**(9), 24343–24373 (2015). http://dx.doi.org/10.3390/s150924343
19. Jirka, S., Bröring, A., Kjeld, P., Maidens, J., Wytzisk, A.: A lightweight approach for the Sensor Observation Service to share environmental data across Europe. Trans. GIS **16**(3), 293–312 (2012). http://dx.doi.org/10.1111/j.1467-9671.2012.01324.x
20. Lagoze, C., Van de Sompel, H.: The open archives initiative: building a low-barrier interoperability framework. In: Proceedings of 1st ACM/IEEE-CS Joint Conference on Digital libraries, JCDL 2001, pp. 54–62. ACM, New York (2001). http://doi.acm.org/10.1145/379437.379449
21. Li, S., Da Xu, L., Zhao, S.: The Internet of Things: a survey. Inf. Syst. Front. **17**(2), 243–259 (2015). http://dx.doi.org/10.1007/s10796-014-9492-7
22. Mulligan, G., Gracanin, D.: A comparison of SOAP and REST implementations of a service based interaction independence middleware framework. In: Proceedings of Winter Simulation Conference (WSC), pp. 1423–1432, December 2009. http://dx.doi.org/10.1109/WSC.2009.5429290
23. Observations and Measurements - XML implementation. Implementation Standard 10–025r1, Open Geospatial Consortium (2011)
24. OGC Sensor Observation Service 2.0. Implementation Standard 12–006, Open Geospatial Consortium (2012)
25. OGC Sensor Observation Service 1.0. Standard 06–009r6, Open Geospatial Consortium (2007)
26. OGC Sensor Observation Service 2.0 Hydrology Profile. Best Practice Paper 14–004r1, Open Geospatial Consortium (2014)
27. SensorML, O.G.C.: Model and XML. Encoding Standard 12–000, Open Geospatial Consortium (2014)
28. OGC SensorThings API Part 1: Sensing. Implementation Standard 15–078r6, Open Geospatial Consortium (2016)
29. OGC Web Feature Service 2.0. Interface Standard 09–025r2, Open Geospatial Consortium (2014)
30. Papazoglou, M., Georgakopoulos, D.: Service-oriented computing. Commun. ACM **46**(10), 25 (2003). https://doi.org/10.1145/944217.944233
31. Perera, C., Zaslavsky, A., Christen, P., Georgakopoulos, D.: Sensing as a service model for smart cities supported by Internet of Things. Trans. Emerg. Telecommun. Technol. **25**(1), 81–93 (2014). https://doi.org/10.1002/ett.2704
32. Pradilla, J., Palau, C., Esteve, M.: SOSLITE: lightweight Sensor Observation Service (SOS) for the Internet of Things (IoT). In: ITU Kaleidoscope: Trust in the Information Society (K-2015), pp. 1–7. IEEE, December 2015. https://doi.org/10.1109/Kaleidoscope.2015.7383625
33. Regueiro, M.A., Viqueira, J.R., Taboada, J.A., Cotos, J.M.: Virtual integration of sensor observation data. Comput. Geosci. **81**, 12–19 (2015). http://dx.doi.org/10.1016/j.cageo.2015.04.006

34. Reitz, K.: Requests: HTTP for humans (2017). http://docs.python-requests.org/en/master/. Accessed 22 Jan 2017
35. Rouached, M., Baccar, S., Abid, M.: RESTful sensor web enablement services for wireless sensor networks. In: IEEE Eighth World Congress on Services, pp. 65–72. IEEE, June 2012. https://doi.org/10.1109/SERVICES.2012.48
36. Samourkasidis, A., Athanasiadis, I.N.: A miniature data repository on a Raspberry Pi. Electronics **6**(1) (2017). http://dx.doi.org/10.3390/electronics6010001
37. The Python standard library: random - Generate pseudo-random numbers. (2017). https://docs.python.org/2/library/random.html. Accessed 22 Jan 2017
38. The Python standard library: time - time access and conversions (2017). Accessed 22 Jan 2017. https://docs.python.org/2/library/sqlite3.html
39. Yazar, D., Dunkels, A.: Efficient application integration in IP-based Sensor networks. In: Proceedings of 1st ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings, BuildSys 2009, pp. 43–48. ACM, New York (2009). http://doi.acm.org/10.1145/1810279.1810289