electronics

MDPI

Article

A Miniature Data Repository on a Raspberry Pi

Argyrios Samourkasidis and Ioannis N. Athanasiadis *

Information Technology Group, Wageningen University, Hollandseweg 1, Wageningen 6706 KN, The Netherlands; argyrios.samourkasidis@wur.nl

* Correspondence: ioannis@athanasiadis.info or ioannis.athanasiadis@wur.nl; Tel.: +31-317-480-166

Academic Editors: Steven J. Johnston and Simon J. Cox

Received: 22 September 2016; Accepted: 15 December 2016; Published: 28 December 2016

Abstract: This work demonstrates a low-cost, miniature data repository proof-of-concept. Such a system needs to be resilient to power and network failures, and expose adequate processing power for persistent, long-term storage. Additional services are required for interoperable data sharing and visualization. We designed and implemented a software tool called Airchive to run on a Raspberry Pi, in order to assemble a data repository for archiving and openly sharing timeseries data. Airchive employs a relational database for storing data and implements two standards for sharing data (namely the Sensor Observation Service by the Open Geospatial Consortium and the Protocol for Metadata Harvesting by the Open Archives Initiative). The system is demonstrated in a realistic indoor air pollution data acquisition scenario in a four-month experiment evaluating its autonomy and robustness under power and network disruptions. A stress test was also conducted to evaluate its performance against concurrent client requests.

Keywords: Raspberry Pi; data repository; interoperability; data archive; data sharing; Sensor Observation Service; Protocol for Metadata Harvesting; indoor air pollution; data acquisition; persistent storage; low cost hardware; Internet of Things

1. Introduction

Raspberry Pi has emerged as a key component in research, education and amateur cyber-physical systems. Raspberry Pi is a low-cost, mini-computer featuring processing, networking and video decoding capabilities [1]. It has no permanent storage; the user may instead attach an SD card. It also exposes General Purpose Input–Output pins (GPIO) to connect with low-level peripheral devices through *Hardware Attached on Top* (HAT). Popular HATs include LEDs, motor controllers, sensors, and GPS devices [2].

Raspberry Pi has been developed primarily with the intention to encourage computer education in schools and the developing world, with the open philosophy in mind, as both the hardware design and operating system are open-licensed. Raspberry Pi has been demonstrated in a variety of applications beyond an educational context, including home-automation systems [3], fire alarm systems [4], home-security [5,6], health supply chains monitoring [7], smart city applications [8–10] and environmental monitoring systems [11]. Tanenbaum et al. [12] viewed Raspberry Pi and similar technologies as enablers for democratizing technology and enabling creativity.

Despite the diversity of Raspberry Pi applications, little research has been done to investigate Raspberry Pi as a performing data repository. The low acquisition cost, the open hardware and software philosophy, and its capacity for interfacing with a variety of peripherals, renders Raspberry Pi a very good candidate for boosting open data, crowd-sourcing and citizen science movements. For instance, Raspberry Pi was employed to create a citizen observatory for water and flood management [13]. Muller et al. [14] discuss its potential use for crowdsourcing applications in climate and atmospheric sciences.

In this work, we present a proof-of-concept that Raspberry Pi can be used as a miniature, low-cost data repository that offers *persistent* data storage, and interoperable data sharing services over the Internet. We demonstrate *Airchive*, a system that stores and serves timeseries data recorded by a HAT equipped with air quality sensors, and investigate the system's robustness against power and network shortages. We also conducted a stress test in order to identify system limitations. The rest of the paper is structured as follows: in Section 2, we study the feasibility of the approach, by reviewing related work. Section 3 presents the overall system architecture, along with user types, system requirements, and key functionality. Section 4 presents the software platform developed and hardware utilized. Section 5 details our experiments with the system and presents the lessons learned, documenting difficulties and incidents arisen during the experiment period. Finally, Section 6 provides a discussion and lays the groundwork for future work. Section 7 provides a conclusion of the research.

2. Related Work

In principle, a data repository needs to offer persistent data storage, along with added-value services, as those for data processing, dissemination and visualization. Such services are similar to those offered by a Wireless Sensor Network (WSN) [15], an area where Raspberry Pi has been thoroughly investigated as a gateway node (or base station). A gateway node is the intermediate among sensor nodes and external networks. Its functionalities are regarded with (a) coordination (e.g., configuration of sensor nodes); (b) data storage; (c) data processing and (d) data dissemination to external clients [16]. Most prominent advances in the usage of Raspberry Pi in WSNs have been done in the domains of (a), (c), and advanced data visualization.

Raspberry Pi has been used as a coordinator in a ZigBee mesh network interfacing with the World Wide Web. In [17], a Raspberry Pi performs as a gateway node and processes observations derived from the sensor nodes, stores them on a local database and provides visualization services to external users.

Data processing on the Raspberry Pi to offline calibrate sensor readings and provide data visualization is presented in [18]. Specifically, a Round Robin Database [19] was used for fast storage of sensor data with a constant disk footprint. This was done by keeping only the recent measurements in high resolution and statistical summaries for older recordings.

Advanced data visualization and image capturing is demonstrated in a volcanic monitoring system based on a Raspberry Pi [20]. The Raspberry Pi creates and communicates graphs through commercial messenger applications—for example Whatsapp—while data are transferred daily to an external system for archival.

From the works above, it becomes clear that a Raspberry Pi may serve as a node that offers data storage, processing and visualization services, while still remaining a coordinating device interfacing sensors with the Internet. In most cases, data are forwarded to a remote, resourceful node in order to be archived in the long term. In this work, we aim to demonstrate that a Raspberry Pi can become an active archiver of its own sensor recordings, and investigate whether it is powerful enough to provide data storage and dissemination services on site.

3. The Airchive System

3.1. Objectives

Airchive [21] is a software product intended for being deployed on a Raspberry Pi to turn it into a self-contained data repository. *Airchive* provides data capture and dissemination services for timeseries measurements. There are two objectives in developing this system.

The first is to investigate long-term storage potential on a Raspberry Pi. The challenge here is inherited by the Raspberry Pi hardware limitations. *Airchive* provides with a persistent storage mechanism that is able to safe-keep its data in a trustworthy manner. We experimented this feature further, considering storage on both SD cards and USB disks attached with the Raspberry Pi.

The second is to demonstrate Raspberry Pi capacity to interoperate at the machine level through standard protocols for data sharing. *Airchive* adopts two mainstream standards to exhibit interoperability at the machine level. The first is the Sensor Observation Service (SOS), the Open Geospatial Consortium (OGC) standard tailored for sharing sensor observations [22]. SOS defines a Web service interface which allows querying observations and metadata of heterogeneous sensor systems. The second is the Protocol for Metadata Harvesting (OAI/PMH), an Open Archives Initiative low-barrier mechanism for repository interoperability [23]. OAI/PMH is a generic protocol for sharing metadata among archives and has been widely adopted by digital libraries. Both SOS and OAI/PMH offer services that are invoked over the HTTP protocol.

3.2. Requirements

Airchive operates as a self-contained, autonomous repository for timeseries data archival and dissemination. It is a technical system that involves both software and hardware components, and needs to comply with certain non-functional requirements. From a software perspective, *Airchive* needs to be built with open-source tools and frameworks and be extensible, in order to respect the philosophy of the Raspberry Pi movement and maximize the potential for future uptake. Hardware support *Airchive* should be low-cost and resilient to power and network shortages. This will allow its use in remote locations, or in the developing world. The overall *Airchive* system should require low-technical skills to install, operate and maintain.

We identified three use cases for the *Airchive* system.

(a) **Web users** access the system through the Internet via a public webpage. They explore current or historical *Airchive* data, and they are interested in graphical representations of the content. Typically, a Web user is able to query for the data stored in *Airchive*, and the system will respond with a graph of the data requested. They may also download data in common formats, such as JSON (JavaScript Object Notation), CSV (Comma-separated values), GeoJSON [24] and GeoRSS [25].

(b) **Software agents** interact with *Airchive* for retrieving data or harvesting metadata. They may use different protocols and vocabularies to submit their requests. One may follow the SOS protocol for retrieving raw timeseries data, while another could use the OAI/PMH to get meta-information of the digital resources stored. Software agents interact with the system with RESTful Web services (Representational state transfer services) [26] over the HTTP protocol.

(c) The **system owner** has full access both locally and from the Internet via Secure Shell (SSH). Her responsibilities are to administer the system by updating system software or restarting the device.

Interoperability is an essential requirement of such a system. *Airchive* offers query services for software agents via SOS and OAI/PMH standards. SOS queries return responses in Extensible Markup Language (XML) using OGC vocabularies (as Observation & Measurements (O&M) [27], or Sensor Model Language (SensorML) [28]). OAI/PMH responses may be encoded in more than one metadata profile, including Dublin Core, a generic purpose metadata schema for annotating digital artifacts [29]. By incorporating a variety of service offerings, we demonstrate the capabilities of a Raspberry Pi to operate with several clients, using different protocols and vocabularies, and support for **syntactic interoperability**.

Software development is based on our previous work reported in [30]. We further improved the software system to host generic timeseries data. The current version has been thoroughly tested and is available as an open source software package [21]. In this version, all of the metadata that are disseminated through OAI/PMH are calculated *on-the-fly* (instead of being stored permanently). This is a design choice to demonstrate the powerful processing power of Raspberry Pi.

Airchive can operate autonomously and with minimal user interventions. In the experiments discussed below, *Airchive* has been operating unattended for four months in order to evaluate its capabilities for *long-term* operation, as reliability, self-recovery and resilience to power and network failures.

3.3. Abstract Architectural Design

Airchive software platform was designed for the Raspberry Pi to turn it into a self-contained station for timeseries data archival and dissemination. It serves both real-time access and long-term storage and retrieval of sensor data, while also offering services for metadata harvesting. *Airchive* follows the *Sensing as a Service* paradigm [31] and is composed of five components that are implemented as loosely-coupled services, rendering the software highly extensible. The abstract architectural design is depicted in Figure 1.

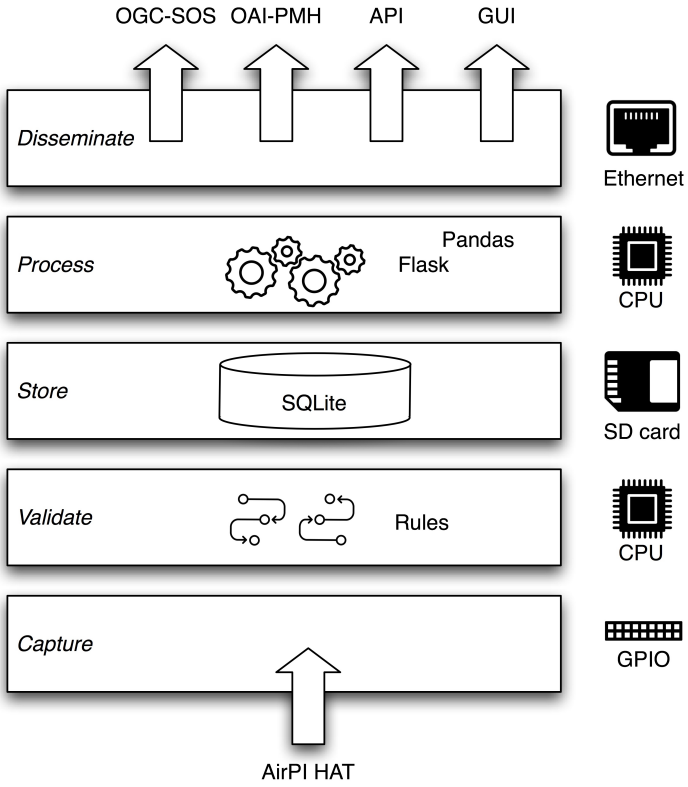

Figure 1. *Airchive* abstract architecture. System services are shown as layered components on the **left**, with relevant technologies. On the **right**, corresponding Raspberry Pi features are illustrated.

The data capture component (optional) comes first that actually collects sensed measurements from one or more sensor devices connected to the Raspberry Pi. This component is custom to hardware and/or sensors used. Our implementation interfaces with the sensors of the AirPi HAT. Nevertheless, the general behavior remains the same: at certain time intervals, it acquires the results from the sensors.

A data validation component (optional) may sit between data capture and data storage components. Its role is to apply quality assurance/quality control process and identify hardware or sensor errors. Additionally, it could associate the measurement with a quality flag by applying rules or more empirical procedures (i.e., statistical, data driven) [32–35]. Such a component is essential for ensuring data reliability and user confidence.

The data storage component permanently stores sensor data in a relational database along with a time stamp. In order to be database-independent, an Object Relational Mapping (ORM) framework was utilized. The data storage component is also responsible for retrieving the data from permanent storage.

The data processing component is an intermediate layer between data storage and Web services. It transforms arguments (submitted by users/harvesters with their queries) into appropriate database queries, using the ORM framework. It also works in the other way around, as it formats database outputs according to user requests, using different formats (i.e., XML, JSON, CSV) or dictionaries

(i.e., O&M, SensorML, Dublin Core). Finally, it offers descriptive statistics calculations *on-the-fly* (e.g., maximum, minimum, rolling mean, average and percentiles).

Last, but not least, the Web services components offer outlets for interaction with users and agents over the Internet. There are four Web service components in the current system, but more could be added in the future: *Web users* browse the repository and submit queries using a Graphical User Interface (GUI). *Software agents* interact with the SOS server, the OAI/PMH endpoint, or the *Airchive*'s own Application Programming Interface (API).

4. Implementation

4.1. Hardware

Airchive was deployed on a Raspberry Pi Model B. This model is equipped with a 700 MHz ARM processor, weights 45 g and has 512 MB of RAM. It is connected to the Internet through an Ethernet controller and features two USB ports. Instead of a hard disk, it uses an SD card. It is also equipped with 26 GPIO pins (General Purpose Input–Output) for interfacing with various peripherals (HATs). The chosen Operation System was Raspbian; a Linux based distribution for Raspberry Pi.

In order to generate data (i.e., actual observations) to be stored on the *Airchive*, we have chosen to use AirPi, a Raspberry Pi sensory HAT. AirPi is an interface board that connects over GPIO pins and is equipped with low cost air quality and weather sensors. It also follows the open hardware philosophy, and can be further extended with other sensors, including a GPS module [36]. It costs roughly 90 USD including the sensors shown in Table 1. AirPi includes a software module that is able to log sensed data on the cloud.

Table 1. AirPi sensors with their respected observed properties.

Sensor Name	Observed Property	Type	Interface
DHT22	Relative humidity, Temperature	Digital	SPI
BMP085	Atmospheric pressure, Temperature	Digital	SPI
MICS-2710	Nitrogen dioxide	Analog	I2C
MICS-5525	Carbon monoxide	Analog	I2C

The overall system hardware is comprised of a Raspberry Pi Model B equipped with the AirPi HAT, an SD card and a USB memory drive, and it was connected to Internet with an ethernet cable, shown in Figure 2.

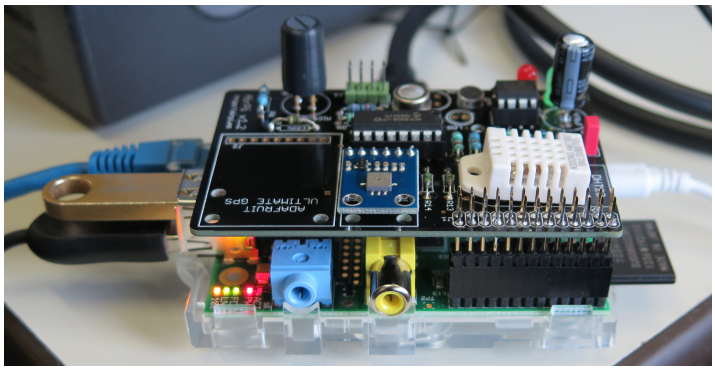

Figure 2. Raspberry Pi Model B with AirPi attached on top.

4.2. Software Development

Airchive software has been developed in Python, and is available as open-source software on github [21] under the GNU Affero General Public License Version 3 [37].

The data capture component interfaces with the AirPi libraries [38] that transform electrical signals into human-understandable values. Data storage employs SQlite [39], an open-source, lightweight relational database for Python and SQLAlchemy library [40] for object-relational mapping. An outline of the data validation component is provided, but not fully implemented, as it is out of the core scope.

The data processing component was developed in three modules. The *Query* module handles client-requested queries and raises appropriate exceptions. Requests must include the sensor, the property and the corresponding timeframe for which the observations will be retrieved. A typical workflow is as depicted in Figure 3. The *Filter* module comprises of a set of statistical filters implemented as Python classes using pandas library [41]. Filters may be instantiated and applied *on-the-fly* on a query result. The *Format* module is responsible for serializing the query results. Jinja2 template engine [42] was used and the formats implemented correspond to the Web services offered. They include XML, GeoRSS, GeoJSON, JSON and CSV formats.

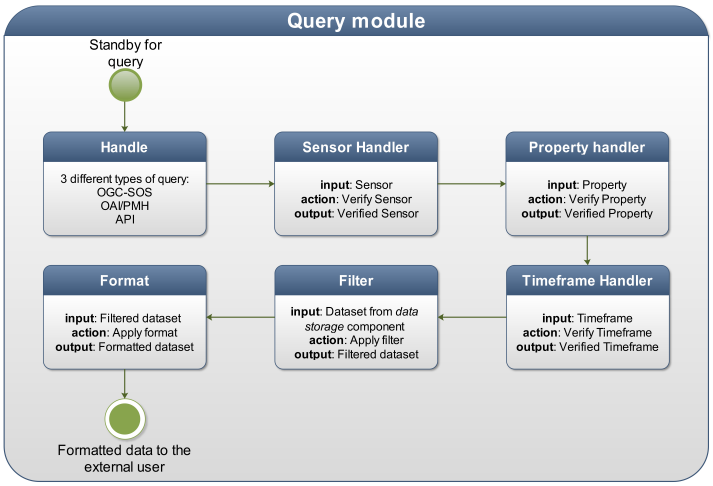

Figure 3. A typical data processing workflow.

The Web service components were developed using the Flask web framework [43], in order to provide clients with static and dynamic content. Flask web framework deploys a web server which responds to `HTTP GET` requests with formatted data. Data requests can be submitted through the three API endpoints that we developed: the *Airchive* own API, SOS and OAI/PMH. A fourth outlet is the *Airchive* GUI, which is meant for the *Web users* to render graphs upon request. It uses the *Airchive* API for getting data, which are subsequently visualized on the client's web browser using Javascript. Graph rendering is facilitated by FLOT [44] and JQuery [45]. Visualizing data occurs on the client browser, which also economizes resources of the Raspberry Pi.

Airchive software is generic in nature, in the sense that is does not require the data capture and validation components, and one could deploy it only with historical data. The system is configured via a file that aligns the timeseries with their semantics, including measured properties and units. The configuration allows for alternative definitions of the same observed properties, which enables the system to serve the same observations with a variety of vocabularies. Currently, we use the definitions of the Semantic Web for Earth and Environmental Terminology (SWEET) [46].

5. Demonstration

5.1. Experimental Design

We deployed the *Airchive* system in a realistic scenario for indoor air quality monitoring. *Airchive* software was installed on a Raspberry Pi equipped with AirPi HAT, and installed indoors, connected to a power supply and the Internet via Ethernet port. We performed two experiments in order to evaluate the system autonomy and robustness, and its performance under load pressure.

5.2. Experiment 1: Autonomy and Robustness

In the first experiment, which lasted for more than 120 days, *Airchive* was exposed to irregular power and network disruptions. We did not interfere in system restoration during the "down" incidents and let the system self-recover.

In this experiment, a moderate sampling frequency to data capture component was set, in order to investigate the system's long-term storage capabilities. A measurement was retrieved from each sensor every 5 min. During this experiment, 183,850 measurements were gradually collected and served. In Table 2, there is a summary of the 24 network outage events observed during this period. Outage events were logged with UptimeRobot [47], a service that monitors web applications and notifies interested parties when an application is not accessible via the Internet. UptimeRobot was used only to log network failures, and did not interfere with our system.

Table 2. Statistics of the 24 the outage events, collected using UptimeRobot.com services.

Metric	Duration
Total downtime	10 days
Median downtime	7 min
Average downtime	543.6 min
St. dev. downtime	2572.4 min

During the first experiment, different users were submitting data queries to the system, in an ad-hoc manner, using the various interfaces: graph visualizations were requested by *Web users*, raw observations by SOS *clients* and derived metadata by OAI/PMH *harvesters*. We did not observe any malfunction for any of the client operations. Current and historical data were monitored, stored and disseminated appropriately, while the automated recovery worked as expected. OAI featured records were calculated *on-the-fly*, upon harvester requests in a timely fashion. We did not observe any notable delays in the capacity of the system to serve its clients.

5.3. Experiment 2: Stress Testing

During the second experiment, we conducted a stress test, in order to provide more insights regarding the system limitations. We investigated the number of concurrent user requests, after which the *Airchive* system delayed to respond. The sampling frequency was increased to 5 s. The experiment lasted for three days, and it collected and served more than 259,000 measurements. We utilized Locust [48], an open source load testing tool written in Python. In Locust, a variable number of clients are deployed to submit concurrent requests to a service. Each Locust client submits a new request only when it receives a response to its previous request.

Locust takes as input the following parameters: (a) the number of concurrent clients; (b) the total number of requests; and (c) a url pointing to the requested resource. We set up three tests. In all cases, clients submitted a hundred requests altogether. The three tests involved the following requests over the Internet, via `HTTP GET`.

In the first test, clients request only the *Airchive* frontpage, which is a static HTML document. No transactions to the database were involved and the response size is constant (8740 bytes). Test 1 verifies that *Airchive* operates properly and examines if pressure on the Web services/dissemination components has an impact on the data capture component.

In the second test, clients request a set of 20 observations using *Airchive*'s API, and the response is formatted as a JSON document. This request requires an SQL query to be submitted to the database, and the response size is 538 bytes. Test 2 corresponds to the use case of a *Web user* that asks for a graph, as *Airchive* transmits the JSON document and the graph is rendered on the client-side.

In the third test, clients ask for the same set of observations as in Test 2, but this time over the SOS protocol, which returns an XML document. This request requires exactly the same SQL query

to be submitted to the database but needs additional formatting for rendering the result in XML. The response size is 16,504 bytes. Test 3 corresponds to the use case of an SOS client asking data from *Airchive*.

We simulated four scenarios, in each of which we deployed a different number of concurrent clients. We tried one, five, 10 and 25 concurrent clients. This is a realistic assumption as the current system is not intended for large-scale deployment. We repeated the process five times for each test and scenario combination and reported two metrics in Table 3: (a) the average response time (ART) in *milliseconds*; and (b) the number of requests served per second (RPS).

Table 3. Experimental results for the three tests and for different numbers of concurrent clients. Average response time (ART) across a hundred requests for each document are reported in milliseconds. System throughput is expressed in requests per second (RPS).

Concurrent Clients	Test 1: Static HTML		Test 2: API/JSON		Test 3: SOS/XML	
	ART (std)	RPS (std)	ART (std)	RPS (std)	ART (std)	RPS (std)
1	66.4 (0.5)	16.2 (0.7)	1830 (27)	0.55 (0.01)	2171 (64)	0.46 (0.01)
5	344 (11)	15.4 (0.6)	9467 (178)	0.52 (0.01)	11,125 (90)	0.44 (0.00)
10	662 (7)	15.4 (0.3)	18,874 (397)	0.51 (0.01)	21,651 (281)	0.44 (0.01)
25	1576 (25)	16.6 (0.7)	45,607 (410)	0.49 (0.01)	49,427 (935)	0.45 (0.01)

Average response time (ART) is a proxy of the average delay to an external user request. For example, a user would have to wait 49.5 s (on average) plus the response time of their submitted request, under the scenario of the 25 concurrent clients for Test 3. As indicated by the results in Table 3, average response time is linearly correlated with the product of (a) number of concurrent users; and (b) average response time achieved when one client submits requests. We verify that requests per second (RPS) depend on the type of the requested document, and is rather stable regardless of the number of concurrent clients.

The introduced overhead to the system response times depends on the request and format type. Requests involving dynamic content are roughly 30 times slower than requests of static content. In the case of dynamic content requests, JSON-formatted responses are served 16% faster than the equivalent in XML.

Interpreting the results, we derive the number of concurrent (human) *Web users* that the system may serve. Assuming that a human user should not wait more than 6 s, we conclude that *Airchive* can serve simultaneously up to two *Web users* of the SOS/XML Web service (Test 3), or three *Web users* of the API/JSON Web service (Test 2). In the case of static content (Test 1), *Airchive* is able to serve up to 82 clients simultaneously. The numbers above do not represent *Airchive*'s maximum capabilities, rather its capacity for serving content to *Web users*.

In contrast, *software agents* interacting with such a system are usually not bound to any time limitation. We conducted further experiments to determine the threshold after which the system started failing to respond to requests. We increased the total number of requests to 500. We started increasing the number of concurrent users by multiples of 5, until requests started to fail. *Airchive* can serve simultaneously, without failure up to 254 (Test 1), 141 (Test 2) and 138 clients (Test 3). In excess of the client numbers above, the system continued to respond with more than one failure. The results are summarized in Table 4. These tests demonstrate *Airchive*'s capacity to work reliably with a significant number of clients.

We underline that despite the heavy workload we introduced during the stress tests, AirPi continued to operate normally. In all cases, we verified with the database content that observations were recorded every 5 s without any loss in all the experiments above (i.e., the data dissemination does not interfere with data capture).

Table 4. Estimates on the number of clients that *Airchive* can serve simultaneously.

User Types	Test 1: Static HTML	Test 2: API/JSON	Test 3: SOS/XML
Human Web users (response in less than 6 s)	82	3	2
Software agents (response guaranteed)	254	141	138

5.4. Incidents and Lessons Learned

During experiment 1, network failures occurred quite often. Those failures, impeded only Web connectivity and apart from the web server, the rest of the *Airchive* components continued to operate properly. We verified that no data loss occurred by cross-checking the time down intervals logged with UptimeRobot with the actual observations stored in the database.

We observed that the system was able to handle power failures, and it self-restored without human intervention. For all 24 outage events during experiment 1, *Airchive* recovered properly by making the Web service available as soon as the Internet connection returned. In this respect, the system demonstrated its persistence and credibility as a repository.

Calculating derived data (metadata) *on-the-fly* provided us with evidence regarding the system's extensibility and enhanced capabilities. Derived metadata, which were disseminated through OAI/PMH, were calculated upon client request. We observed that data were transmitted as fast as if they had been stored in the system. In addition, utilizing a Javascript framework for rendering graphs upon user request added no extra performance overhead to the Raspberry Pi. We did extensively evaluate these features with stress tests in experiment 2, and our experience was that the system performed as expected.

During experiment setup, we stumbled upon a recurring security incident. Given that Raspberry Pi was constantly connected to the Internet, it attracted malicious users after its first boot. We experienced a brute force attack to the SSH protocol that was trying to get unauthorized access to the device. We toughened up *Airchive* with a dedicated security software solution (fail2ban [49]), which prevented any further security incidents of that kind.

Another lesson learned had to do with a potential issue that may arise when power and networks fail at the same time. Raspberry Pi lacks a Real-Time built-in Clock (RTC), and it synchronizes its system clock through the Internet. In the case that an Internet connection is not available upon system boot, the Raspberry Pi system time is misconfigured. In the general use case of *Airchive*, this will not be a problem, but, in our experiments, this will result in errors in the data capture component, which will assign incorrect timestamps to data sensed from the HAT. This problem can be overcome so that the data capture component retrospectively reviews these timestamps when the Internet becomes available. An RTC HAT can be purchased and applied to Raspberry Pi. However, this option increases the total cost.

Last but not least, during the setup phase, we experimented with booting Raspbian and running *Airchive* from the USB disk instead of the SD card. First of all, this is a task that requires advanced technical skills and is still an experimental option not endorsed by the Raspberry Pi makers, and performance is not guaranteed. USB disks provide a cheaper storage option but are prone to failure. We experimented with this option for one month, during which the filesystem was corrupted twice, requiring the operating system and *Airchive* to be re-installed. Observed data were not permanently lost, but their retrieval required technical skills. In contrast, no such incidents occurred when the system operated on an SD card for a much longer period.

6. Discussion

Data persistence is a prerequisite for a data repository. In most efforts made with a Raspberry Pi and reported in the literature in the WSN context, data were periodically backed up in an external

device and were not permanently stored on the embedded device. In the work presented here, *Airchive* relied solely on Raspberry Pi for permanent data storage. Our four-month experiments demonstrated that a Raspberry Pi equipped with an SD card can handle moderate and extensive read/write cycles without any issue; the resilience of SD cards is constantly evolving [50].

The processing capabilities of Raspberry Pi have been investigated in the light of several applications. In *Airchive*, we studied its capacity to calculate and disseminate added-value data and indexes *on-the-fly*, i.e., upon user request. This way, less data are permanently stored and less write cycles are performed, which puts less pressure on SD card life.

Self-restoration from failures is another attribute of WSNs [16], which is also applicable in our work. Self-restoration contributes towards diminishing the technical skills that *Airchive* system owner should possess. During the experiments, the system self-restored from all power and network shortages that have been triggered, demonstrating that after its installation, the system can operate autonomously and without assistance.

We also consider that the *Airchive* system presented here also indirectly contributes to the *open data* movement, especially for the developing world. Besides the low acquisition cost and the low-technical skills required for its deployment, the system by-design responds to the "weak enabling environment" of the developing countries, i.e., intermittent, opportunistic Internet connection. In the frame of this work, we did not demonstrate the system in such conditions. However, we demonstrated that is able to attend to network and power failures.

Security and privacy are also two important attributes of a data repository system, and lay the foundation for future work. The *brute force attack* incident that occurred during the experiments is an illustration of the potential dangers. In addition, given that a data repository system may host personal and/or confidential data, more research should be focused on addressing privacy issues. There is a lack of any authentication mechanism, even in well-established, data dissemination protocols, such as OGC/SOS and OAI/PMH. An authentication mechanism can ensure privacy, and such issues should be addressed in the light of interoperable data dissemination on the application layer.

7. Conclusions

To summarize, we provided a proof-of-concept that current low-cost hardware is reliable enough to boost the *open data* movement. We demonstrated that a Raspberry Pi accompanied with an appropriate software can support persistent data storage, and provide added-value services on site. We designed and implemented an open-source, highly-extensible data repository software, called *Airchive*, to support data visualization, and interoperable data dissemination. We adopted two well-established data dissemination protocols: OGC Sensor Observation Service and Open Archive Initiative/Protocol Metadata Harvesting. Finally, we demonstrated its long-term data storage capabilities and resilience under harsh conditions of power and/or network failures, which take place irregularly. The load testing experiments provided us with insights about the Raspberry Pi performance under simultaneous requests from concurrent external clients.

Supplementary Materials: *Airchive* software is available on github: https://www.github.com/ecologismico/airchive. *Airchive* uptime statistics are available on Zenodo: http://doi.org/10.5281/zenodo.167318. *Airchive* stress test results are available on Zenodo: http://doi.org/10.5281/zenodo.167319. The locust configuration used for the *Airchive* stress test is available on Zenodo: http://doi.org/10.5281/zenodo.167326.

Acknowledgments: Icons in Figure 1 are licensed by Creative Commons CC/BY via the Noun Project, the ethernet port by Michael Wohlwen, the CPU by iconsmind.com, the SD Card by Lemon Liu, improvement by Tomas Knopp, and process, by CBi icons.

Author Contributions: A.S. and I.N.A. conceived and designed the system; A.S. developed the software and performed the experiments; A.S. and I.N.A. analyzed the data; A.S. and I.N.A. wrote the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Upton, E.; Halfacree, G. *Raspberry Pi User Guide*; John Wiley & Sons: Hoboken, NJ, USA, 2014.
2. Nuttall, B. Top 10 Raspberry Pi Add-on Boards. 2016. Available online: https://opensource.com/life/16/7/top-10-Raspberry-Pi-boards (accessed on 12 December 2016).
3. Vujović, V.; Maksimović, M. Raspberry Pi as a Sensor Web node for home automation. *Comput. Electr. Eng.* **2015**, *44*, 153–171.
4. Bahrudin, B.; Saifudaullah, M.; Abu Kassim, R.; Buniyamin, N. Development of Fire Alarm System using Raspberry Pi and Arduino Uno. In Proceedings of the International Conference on Electrical, Electronics and System Engineering (ICEESE), Kuala Lumpur, Malaysia, 4–5 December 2013; pp. 43–48.
5. Sapes, J.; Solsona, F. FingerScanner: Embedding a Fingerprint Scanner in a Raspberry Pi. *Sensors* **2016**, *16*, 220.
6. Chowdhury, M.N.; Nooman, M.S.; Sarker, S. Access Control of Door and Home Security by Raspberry Pi Through Internet. *Int. J. Sci. Eng. Res.* **2013**, *4*, 550–558.
7. Schön, A.; Streit-Juotsa, L.; Schumann-Bölsche, D. Raspberry Pi and Sensor Networking for African Health Supply Chains. In Proceedings of the 6th International Conference on Operations and Supply Chain Management, Bali, Indonesia, 10–12 December 2014.
8. Jung, M.; Weidinger, J.; Kastner, W.; Olivieri, A. Building automation and smart cities: An integration approach based on a service-oriented architecture. In Proceedings of the 27th International Conference on Advanced Information Networking and Applications Workshops (WAINA), Barcelona, Spain, 25–28 March 2013; pp. 1361–1367.
9. Leccese, F.; Cagnetti, M.; Trinca, D. A Smart City Application: A Fully Controlled Street Lighting Isle Based on Raspberry-Pi Card, a ZigBee Sensor Network and WiMAX. *Sensors* **2014**, *14*, 24408–24424.
10. Cagnetti, M.; Leccese, F.; Trinca, D. A New Remote and Automated Control System for the Vineyard Hail Protection Based on ZigBee Sensors, Raspberry-Pi Electronic Card and WiMAX. *J. Agric. Sci. Technol. B* **2013**, *3*, 853.
11. Nikhade, S.G. Wireless sensor network system using Raspberry Pi and ZigBee for environmental monitoring applications. In Proceedings of the International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM), Chennai, India, 6–8 May 2015; pp. 376–381.
12. Tanenbaum, J.; Williams, A.; Desjardins, A.; Tanenbaum, K. Democratizing technology: Pleasure, utility and expressiveness in DIY and maker practice. In Proceedings of the SIGCHI Conf Human Factors in Computing Systems, Paris, France, 27 April–2 May 2013; pp. 2603–2612.
13. Lanfranchi, V.; Ireson, N.; When, U.; Wrigley, S.; Fabio, C. Citizens' observatories for situation awareness in flooding. In Proceedings of the 11th International Conference on Information Systems for Crisis Response and Management (ISCRAM), Pennsylvania State University, University Park, PA, USA, 17 May 2014; pp. 145–154.
14. Muller, C.; Chapman, L.; Johnston, S.; Kidd, C.; Illingworth, S.; Foody, G.; Overeem, A.; Leigh, R. Crowdsourcing for climate and atmospheric sciences: Current status and future potential. *Int. J. Climatol.* **2015**, *35*, 3185–3203.
15. Chang, F.C.; Huang, H.C. A survey on intelligent sensor network and its application. *J. Netw. Intell.* **2016**, *1*, 1–15.
16. Dargie, W.; Poellabauer, C. *Fundamentals of Wireless Sensor Networks: Theory and Practice*; John Wiley & Sons: Hoboken, NJ, USA, 2010.
17. Ferdoush, S.; Li, X. Wireless sensor network system design using Raspberry Pi and Arduino for environmental monitoring applications. *Procedia Comput. Sci.* **2014**, *34*, 103–110.
18. Lewis, A.; Campbell, M.; Stavroulakis, P. Performance evaluation of a cheap, open source, digital environmental monitor based on the Raspberry Pi. *Measurement* **2016**, *87*, 228–235.
19. Oetiker, T. RRDtool. 2014. Available online: http://oss.oetiker.ch/rrdtool/ (accessed on 12 December 2016).
20. Moure, D.; Torres, P.; Casas, B.; Toma, D.; Blanco, M.J.; Del Río, J.; Manuel, A. Use of Low-Cost Acquisition Systems with an Embedded Linux Device for Volcanic Monitoring. *Sensors* **2015**, *15*, 20436–20462.
21. Samourkasidis, A.; Athanasiadis, I.N. Airchive Software. 2016. Available online: https://github.com/ecologismico/airchive (accessed on 12 December 2016).

22. Open Geospatial Consortium. *OGC Sensor Observation Service 2.0*; Implementation Standard 12-006; Open Geospatial Consortium: Wayland, MA, USA, 2012.
23. Lagoze, C.; Van de Sompel, H. The Open Archives Initiative: Building a low-barrier interoperability framework. In *Proceedings of the 1st ACM/IEEE-CS Joint Conference on Digital Libraries, Roanoke, VA, USA, 24–28 June 2001*; ACM: New York, NY, USA, 2001; pp. 54–62.
24. Butler, H.; Daly, M.; Doyle, A.; Gillies, S.; Hagen, S.; Schaub, T. *The GeoJSON Format*; RFC 7946; The Internet Engineering Task Force: 2016, Available online: http://www.rfc-editor.org/info/rfc7946 (accessed on 12 December 2016).
25. GeoRSS: Geographically Encoded Objects for RSS Feeds. 2014. Available online: http://www.georss.org (accessed on 12 December 2016).
26. Richardson, L.; Ruby, S. *RESTful Web Services*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2008.
27. Open Geospatial Consortium. *Observations and Measurements—XML Implementation*; Implementation Standard 10-025r1; Open Geospatial Consortium: Wayland, MA, USA, 2011.
28. Open Geospatial Consortium. *OGC SensorML: Model and XML*; Encoding Standard 12-000; Open Geospatial Consortium: Wayland, MA, USA, 2014.
29. Dublin Core Metadata Initiative (DCMI) Metadata Terms. Available online: http://dublincore.org/documents/dcmi-terms/ (accessed on 12 December 2016).
30. Samourkasidis, A.; Athanasiadis, I.N. Towards a low-cost, full-service air quality data archival system. In Proceedings of the 7th International Congress on Environmental Modelling and Software, International Environmental Modelling and Software Society (iEMSs), San Diego, CA, USA, 15–19 June 2014.
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.* **2014**, *25*, 81–93.
32. Athanasiadis, I.N.; Mitkas, P.A. Knowledge discovery for operational decision support in air quality management. *J. Environ. Inform.* **2007**, *9*, 100–107.
33. Athanasiadis, I.N.; Milis, M.; Mitkas, P.A.; Michaelides, S.C. A multi-agent system for meteorological radar data management and decision support. *Environ. Model. Softw.* **2009**, *24*, 1264–1273.
34. Athanasiadis, I.; Rizzoli, A.; Beard, D. Data Mining Methods for Quality Assurance in an Environmental Monitoring Network. In *Proceedings of the 20th International Conference on Artificial Neural Networks (ICANN 2010), Thessaloniki, Greece, 15–18 September 2010*; Lecture Notes in Computer Science; Springer: Thessaloniki, Greece, 2010; Volume 6354; pp. 451–456.
35. Athanasiadis, I.N.; Mitkas, P.A. An agent-based intelligent environmental monitoring system. *Manag. Environ. Qual.* **2004**, *15*, 238–249.
36. Dayan, A.; Hartley, T. AirPi. 2013. Available online: http://airpi.es (accessed on 12 December 2016).
37. Free Software Foundation. GNU Affero General Public License. 2007. Available online: https://www.gnu.org/licenses/agpl.html (accessed on 12 December 2016).
38. Hartley, T. AirPi Software. 2013. Available online: https://github.com/tomhartley/AirPi (accessed on 12 December 2016).
39. sqlite3—DB-API 2.0 Interface for SQLite Databases. 2016. Available online: https://docs.python.org/2/library/sqlite3.html (accessed on 12 December 2016).
40. Bayer, M. SQLAlchemy: The Python SQL Toolkit and Object Relational Mapper. 2016. Available online: http://www.sqlalchemy.org (accessed on 12 December 2016).
41. McKinney, W. pandas: A Foundational Python Library for Data Analysis and Statistics. In Proceedings of the Workshop Python for High Performance and Scientific Computing (SC11), Seattle, WA, USA, 18 November 2011.
42. Ronacher, A. Jinja2. 2008. Available online: http://jinja.pocoo.org (accessed on 12 December 2016).
43. Ronacher, A. Flask. 2010. Available online: http://flask.pocoo.org (accessed on 12 December 2016).
44. Laursen, O.; Schnur, D. Flot: Attractive JavaScript Plotting for jQuery. 2007. Available online: http://www.flotcharts.org (accessed on 12 December 2016).
45. jQuery. 2016. Available online: https://jquery.com (accessed on 12 December 2016).
46. Raskin, R.G.; Pan, M.J. Knowledge representation in the semantic web for Earth and environmental terminology (SWEET). *Comput. Geosci.* **2005**, *31*, 1119–1125.
47. UptimeRobot. 2016. Available online: https://uptimerobot.com (accessed on 12 December 2016).

48. Heyman, J.; Hamrén, J.; Byström, C.; Heyman, H. Locust: An Open Source Load Testing Tool. 2011. Available online: http://locust.io (accessed on 12 December 2016).
49. Sumsal, F.; Brester, S.G.; Szépe, V.; Halchenko, Y. Fail2ban. 2005. Available online: http://www.fail2ban.org/ (accessed on 12 December 2016).
50. SD Association. 2016. Available online: https://www.sdcard.org (accessed on 12 December 2016).