Data Redundancy Management in Connected Environments

Elio Mansour
Univ. Pau & Pays Adour, E2S UPPA, LIUPPA
Anglet, 64600, France
elio.mansour@univ-pau.fr

Faisal Shahzad
Univ. Pau & Pays Adour, E2S UPPA, LIUPPA
Anglet, 64600, France
faisal.shahzad@univ-pau.fr

Joe Tekli
Lebanese American University, E.C.E. Dept.
36 Byblos, Lebanon
joe.tekli@lau.edu.lb

Richard Chbeir
Univ. Pau & Pays Adour, E2S UPPA, LIUPPA
Anglet, 64600, France
richard.chbeir@univ-pau.fr

ABSTRACT

Connected environments are typically defined as physical infrastructures (e.g., building) equipped with sensors that produce and exchange raw data. Although the sensed data is considered to contain useful and valuable information, yet it might include various inconsistencies such as data redundancies, anomalies, and missing values. In this work, we focus on managing sensor data redundancies in connected environments. Existing works often suffer from (i) disregarding either network core or edge device redundancies; (ii) disregarding the limited capabilities of edge devices; and (iii) disregarding sensors mobility and the dynamicity of the network. To address these limitations, we propose a framework for data redundancy management at the device level, denoted DRMF. We describe its modules, and clustering-based algorithms. Moreover, our proposal detects temporal, and spatial-temporal redundancies in order to consider both static and mobile devices/sensors. Finally, we present our experimental protocol and share preliminary results.

CCS CONCEPTS

• **Information systems → Data cleaning**; • **Computer systems organization → Sensor networks**.

KEYWORDS

Connected Environments, Internet of Things, Data Redundancy

ACM Reference Format:
Elio Mansour, Faisal Shahzad, Joe Tekli, and Richard Chbeir. 2020. Data Redundancy Management in Connected Environments. In *16th ACM Symposium on QoS and Security for Wireless and Mobile Networks (Q2SWinet'20), November 16–20, 2020, Alicante, Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3416013.3426451

1 INTRODUCTION

Recent advances in data management and sensing technologies have allowed physical infrastructures (e.g., buildings, homes, and cities) to become more connected. Using sensor networks, these connected environments produce huge amounts of sensed data that can be exploited for various high-level applications (e.g., environment monitoring, event detection, and energy management). Although the sensed data is considered to contain useful and valuable information, pre-processing is still needed in most cases since the observations are in raw form and often suffer from various inconsistencies [3, 10] (e.g., redundancies, anomalies, and missing values). In this work, we focus on handling sensor data redundancies in connected environments. Removing unnecessarily redundant data is pivotal since it (i) allows improved querying over the gathered data [7]; (ii) reduces the communication costs and resource consumption of edge devices that share the data regularly; and (iii) provides AI-based services with cleaned and ready-to-use data sets for their underlying advanced algorithms. Existing works [2, 4, 6, 8, 9] target data redundancy in connected environments, however they suffer from the following limitations:

(1) *Disregarding either network core or edge device redundancies:* one should be able to query the edge and the core of the network (e.g., to detect the redundancies produced by a device, and aggregate the readings from multiple devices in the environment). Therefore, it is important to handle redundancies not only at the base station level, but also at the device level.
(2) *Disregarding the limited resources of edge devices:* devices at the edge of the network often have limited resources (e.g., processing, memory, and power). Moreover, devices need to exchange data as well as push data to the core. Therefore, it is important not to deplete device resources by excessive communications and heavy processing of redundant data.
(3) *Disregarding environment dynamicity:* dynamic environments include mobile devices/sensors in addition to static nodes. Considering mobile devices allows the detection of new redundancies generated by device mobility.

In this paper, we propose to handle data redundancy in connected environments at the device level. We consider both static and mobile devices that embed sensors, monitor the generated data, and detect redundancies based on temporal and spatial features. Eliminating redundancies at the device level would improve querying performance in the entire network, and reduce costly computations/communications while considering user needs (e.g., querying the edge and core of the network), device needs (e.g., avoiding excessive resource usage), and external services (e.g., enabling the execution of advanced processing and mining services on the data).

To do so, we address the detection of temporal and spatial-temporal redundancies which affect static and mobile devices respectively. We introduce the formal definitions and develop the needed algorithms to handle both types of redundancies. Preliminary evaluation and tests results highlight the potential of our proposal.
In the following, we describe our motivating scenario in Section 2. Then, we compare the related works in Section 3. We detail the framework of our proposal, and present our algorithms for redundancy detection at device level in Section 4. The experimental protocol and preliminary results are provided in Section 5. Finally, Section 6 concludes the paper and discusses future works.

2 MOTIVATING SCENARIO

Consider the following scenario that illustrates a section of a smart parking. Please note that this example does not summarize all data redundancy issues in a connected environment, and is used to highlight the main needs and challenges related to this work. Figure 1 illustrates three parking spaces each containing a static device, and two mobile devices moving in the parking: a mobile phone, and a vehicle. Each device embeds one or more sensors, is equipped with a local memory for temporary storage of observations, and is capable of processing queries and exchanging data with neighbouring devices. Moreover, all devices can push data into the central database. For the sake of brevity, we only consider three types of sensor observations in this example: temperature, occupancy, and CO_2, sensed periodically by the devices' sensors. Finally, the parking manager uses the aforementioned data to monitor specific events, e.g., bad air quality, fires, free parking spaces. To do so, the parking manager has the following needs:

- Need 1. *Considering redundancies at the core and edge:* for querying the parking devices to monitor sensor breakdowns, and anomalies and the central database to retrieve aggregated data, and apply data processing and mining services.
- Need 2. *Considering the limited resources of edge devices:* for having an efficient and low cost inter-device data exchange in order to combine and retrieve location-based information from the parking.
- Need 3. *Considering environment dynamicity:* for querying mobile devices while considering spatial-temporal features.

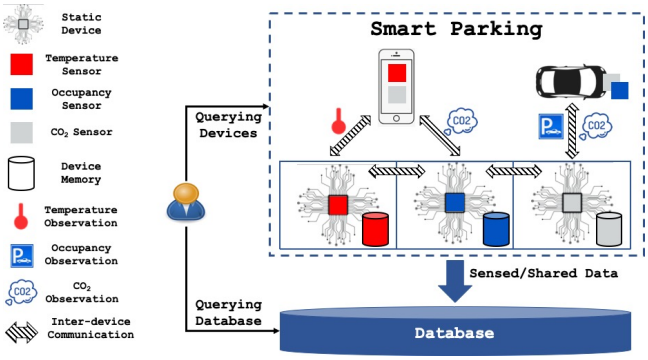

Figure 1: The Smart Parking

As a result, the static and mobile sensors in the smart parking environment will be producing large amounts of data, exchanging some of them among each other, and sending them periodically to the database even if no significant changes occur in the sensed data. For instance, if a car is parked for hours in the same spot, redundant occupancy data is sent during all the occupancy time. Similarly, redundant CO_2 and temperature data will be periodically produced, exchanged, and stored even when the parking is not witnessing any activity (e.g., cars movements, people walking). The large amount of sensor data produced and exchanged in the smart parking environment highlight various challenges from user, device, and dynamicity perspectives:

- User Perspective: How to run simple queries on static/mobile devices and retrieve data efficiently without over-consuming the devices' limited resources? How to run aggregate queries, or advanced mining services on the central database without causing latency issues?
- Device Perspective: How to minimize unnecessary data exchanges between devices to avoid wasting power, processing, memory, and network resources?
- Dynamicity Perspective: How to detect dynamic spatial-temporal redundancies generated by device mobility?

In this work, we tackle the data redundancy problem at the device level in order to address the aforementioned challenges. Before detailing the proposal, we first present and compare some existing approaches for data redundancy management in sensor networks.

3 RELATED WORKS

To compare existing approaches, we propose the following criteria based on the challenges and limitations discussed in Section 2:

- Criterion 1. *Core & edge redundancy consideration:* stating if the approach handles data redundancy at the device and database levels. This enables efficient querying on the edge and core of the network (cf. Need 1).
- Criterion 2. *Edge device resource consideration:* denoting if the approach considers the limited resources of edge devices when processing and exchanging data (cf. Need 2).
- Criterion 3. *Dynamicity consideration:* specifying if the approach considers dynamic redundancies due to device mobility (cf. Need 3).

We review next some of the existing approaches on data redundancy in connected environments.

3.1 Existing Approaches

The authors in [4] present a data reduction scheme for Internet of Things (IoT) using data filtering and fusion. Their approach handles redundancies at the device layer before forwarding non-redundant data to sink nodes. Redundancy detection is solely based on data value deviations. Although, this work handles redundancies at the edge of the network, it does not cover redundant data from mobile devices (cf. Criterion 3). Therefore, specific spatial-temporal redundancies at device level are not handled. In [8], the authors address data redundancies at the core of the network using a supervised machine learning solution based on Support Vector Machine (SVM). They build an aggregation tree for the given size of the network and then apply SVM to recognize data redundancies. In this work, the authors target temporal and spatial redundancies once the data is consolidated in a central node, which provides a redundancy-free

data repository that could be mined using advanced data processing techniques (cf. Criterion 1). However, redundancies are not handled at the device level, and data exchange between devices at the edge remains costly due to unnecessary communications. Moreover, the authors do not consider spatial-temporal redundancies generated by mobile devices (cf. Criterion 2 and 3). In [2], the authors focus on the spatial distribution of sensors in the environment, and how it can be managed in order to prevent redundancies. To do so, a graph of nodes and detected events is constructed from raw sensory data to identify nodes producing redundant data. Next, these so-called "redundant" nodes are either relocated or put into sleep mode using a circle packing technique to enhance coverage while minimizing energy usage during relocation. This work only handles redundancy from a sensor deployment perspective (i.e., avoiding deploying sensors that provide the same type of data in the same area). Therefore, the emphasis is on detecting redundant sensor nodes and not the data itself. Moreover, the proposal does not consider sensor mobility (cf. Criterion 3). In [9], the authors present a data de-duplication technique in healthcare-based Internet of Things (IoT). They propose a Controlled Window-size based Chunking Algorithm (CWCA) to identify cut-points in sensor data distributions. The data de-duplication is applied at the collector node (i.e., at the core and does not consider edge node redundancies). More recently, the authors in [6] propose a data redundancy elimination technique using an unsupervised learning approach based on data clustering. The authors suggest clustering the edge nodes based on their produced sensory data in order to aggregate identical data to eliminate redundancies, before storing the data in the cloud. However, these works [6, 9] do not consider device mobility and spatial-temporal redundancies (cf. Criterion 3).

3.2 Comparison Summary

Table 1 shows that none of the aforementioned works cover all the required criteria addressed in our present study. Most approaches focus on handling redundancies at the core of the network, thus neglecting the impact of redundancies on the edge devices where resources are often limited (e.g., power, processing, and memory).

	Criterion 1 *Core & Edge*	Criterion 2 *Edge Resources*	Criterion 3 *Dynamicity*
Chowdhury S. et al. [2]	×	×	×
Ismael W. et al. [4]	✓	✓	×
Li S. et al. [6]	✓	✓	×
Patil P. et al. [8]	×	×	×
Ullah A. et al. [9]	×	×	×

Table 1: Related Works Recap

4 PROPOSAL

To address the limitations identified in the previous section, we introduce DRMF, a Data Redundancy Management Framework. In this study, we show how DRMF handles sensor data redundancy at the edge device level, considering both static and mobile devices, in order to eliminate redundancies from the source before reaching the core of the network. The DRMF overall architecture is depicted in Figure 2. It consists of two main modules: (i) datatype filtering which separates the input data into type-based data collections; and (ii) redundancy cleaning which detects then cleans temporal or spatial-temporal redundancies from each data collection. In the following subsections, we start by describing the nature of sensory data in a dynamic environment. Then, we detail the redundancy management at device level. Finally, we focus on the redundancy checker sub-module in order to detail the proposed redundancy detection algorithms.

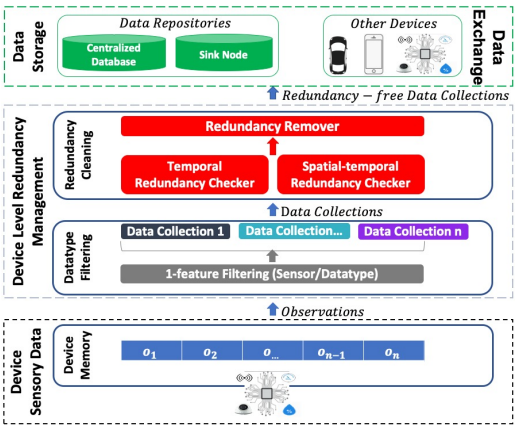

Figure 2: The DRMF Architecture

4.1 Sensory Data in a Dynamic Environment

Connected environments contain diverse devices each embedding one or more sensors that provide data from the real world. Static devices are immobile, therefore the data generated by such devices could be redundant temporally. However, mobile devices move around in the environment while producing data. This potentially generates spatial-temporal redundancies. In the following, we provide a set of formal definitions that allow us to describe data items and both temporal and spatial dimensions (cf. Criterion 3).

Definition 1 (Data Items)**.** We formally define a data item d as a 5-tuple:

$$d : \langle a, v, t, l, s \rangle \quad where: \tag{1}$$

- a is the data attribute
- v is the data value
- t is the creation temporal stamp of d (cf. Definition 2)
- l is the creation location stamp of d (cf. Definition 3)
- s is the data source that produced/created d ■

Definition 2 (Temporal Stamp Definition)**.** A temporal stamp t designates a single discrete temporal value formally defined as a 2-tuple:

$$t = \langle format, value \rangle \quad where: \tag{2}$$

- $format$ is a string indicating the format of the date-time value of t (e.g., "dd-MM-yyyy hh:mm:ss")
- $value$ is the timestamp value (e.g., 10-11-2020 15:34:23 following the sample time format mentioned above) ■

Definition 3 (Location Stamp Definition)**.** A location stamp l is s a discrete and instantaneous location value defined as a 2-tuple:

$$l = \langle format, value \rangle \quad where: \tag{3}$$

- $format$ is the location referential format following which the location stamp value will be represented (e.g., default GPS, or Cartesian, Spherical, Cylindrical)
- $value = \langle x, y, z \rangle$ is a discrete and instantaneous value, where x, y, and z designate individual coordinate values (the coordinates can be translated into the referential of choice following the designated format) ■

Table 2 shows an excerpt of the data produced by a device having two embedded sensors S1, and S2 that produce CO_2 and temperature observations respectively.

a	v	t		l				s
		format	value	format	value			
					x	y	z	
CO_2	98	dd/mm/yyyy hh:mm:ss	10/02/2019 10:00:00	cartesian	8	12	8	S1
CO_2	109	dd/mm/yyyy hh:mm:ss	10/02/2019 10:02:00	cartesian	6	8	6	S1
CO_2	110	dd/mm/yyyy hh:mm:ss	10/02/2019 10:04:00	cartesian	2	4	8	S1
CO_2	111	dd/mm/yyyy hh:mm:ss	10/02/2019 10:06:00	cartesian	4	6	4	S1
Temperature	22	dd/mm/yyyy hh:mm:ss	10/02/2019 10:08:00	cartesian	6	4	8	S2

Table 2: Data Items Example

4.2 Redundancy Management

In a typical connected environment, the sensor observations are temporarily stored in the device's memory, before the data is eventually transmitted to a permanent storage repository or another device. In DRMF, we propose to detect and handle redundancies prior to data storage or transmission. As a result, our redundancy management process consists of two steps: (i) datatype filtering; and (ii) redundancy cleaning.

4.2.1 Step 1: Datatype Filtering. Since the device could embed various sensors, the internal memory could contain different datatypes (i.e., different data attributes or features). The example provided in Table 2 shows two different datatypes: CO_2, and temperature. Therefore in order to detect redundancies in the data stored locally on the device, one starts by filtering the data into collections having the same attributes (or datatypes), hence the datatype filtering module. In this step, the data is split into separate collections based on the datatype. In the following step, we detect redundancies within each data collection (cf. Figure 2). To illustrate the datatype filtering process, the data shown in Table 2 produces two distinct data collections: the first for CO_2 data (first four tuples); and the second for temperature data containing the last tuple.

4.2.2 Step 2: Redundancy Cleaning. In this step, the redundancy checker applies our redundancy detection algorithms over the sensor's locally stored data. More specifically, the aforementioned algorithms cluster the data based on the deviation of the data item values, while also considering the temporal, or spatial-temporal spread (or coverage) of the clusters (i.e., sets of redundant data). We propose to detect redundancies using an unsupervised cluster-based approach for two main reasons: (i) to avoid applying supervised learning which requires training time and computation power on the edge where resources are limited; and (ii) since training data for supervised learning algorithms might not be available at the device level. We provide two redundancy checking algorithms: one for temporal redundancy detection, specifically designed to handle data from static devices; and another for spatial-temporal redundancy detection, specifically designed to handle data from mobile devices. A temporal redundancy (cf. Definition 4) is defined as a cluster of redundant values spanning over a specific time coverage (cf. Definition 5). Similarly, a spatial-temporal redundancy (cf. Definition 6) is defined as a cluster of redundant values spanning over a specific time coverage, and spatial coverage (cf. Definition 7).

Definition 4 (Temporal Redundancy)**.** A temporal redundancy tr is defined as a 2-tuple:

$$tr : (coverage_t, D) \;\; where: \tag{4}$$

- $coverage_t$ is the temporal coverage during which the data is temporally redundant
- $D = \bigcup_{j=0}^{z} d_j$ is a cluster of redundant data items where:
  - $\forall\ d_j \in D,\ d_j.t \in coverage_t.\delta_t$
  - $\forall\ d_{j1}, d_{j2} \in D,\ d_{j1}.a = d_{j2}.a$
  - $\forall\ k \in \mathbb{N}^+,\ d_k.v = d_{centroid}.v \pm \delta_v \quad where:$
    - $d_{centroid}.v$ is the centroid value of all data items in D
    - δ_v is an acceptable deviation threshold

Remark. The threshold δ_v is calculated based on the data distribution within the redundant data set D. ■

Definition 5 (Temporal Coverage Definition)**.** A temporal coverage $coverage_t$ is a time interval consisting of an ordered collection of temporal stamps enclosed within a start and an end stamp, describing the temporal coverage of a sensor observation (e.g., video feed) or a group of observations (e.g., scalar measurements, images). Formally, it is defined as a 2-tuple:

$$coverage_t = \langle \delta_t, g_t \rangle \quad where: \tag{5}$$

- $\delta_t = [t_s, t_e]$ is a temporal interval where:
  - $t_s < t_e$ is the start temporal stamp
  - t_e is the end temporal stamp
- g_t is a temporal granularity or unit of the temporal coverage (e.g., millisecond, second, minute, etc.) ■

Definition 6 (Spatio-Temporal Redundancy)**.** A spatio-temporal redundancy str is defined as a 3-tuple:

$$str : (coverage_t, coverage_l, D) \;\; where: \tag{6}$$

- $coverage_t$ is the temporal coverage
- $coverage_l$ is the location coverage
- $D = \bigcup_{j=0}^{z} d_j$ is a cluster of redundant data where:
  - $\forall\ d_j \in D,\ d_j.t \in coverage_t.\delta_t$
  - $\forall\ d_j \in D,\ d_j.l \in coverage_l.\delta_l$
  - $\forall\ d_{j1}, d_{j2} \in D,\ d_{j1}.a = d_{j2}.a$
  - $\forall\ k \in \mathbb{N}^+,\ d_k.v = d_{centroid}.v \pm \delta_v \quad where:$
    - $d_{centroid}.v$ is the centroid value of all data items in D
    - δ_v is an acceptable deviation threshold ■

Definition 7 (Location Coverage Definition)**.** A location coverage $coverage_l$ is the set of spatial stamps designating the surface coverage in which a sensor observation is created (e.g., area in which a video stream or a bunch of mobile measurements are recorded). Formally, it is defined as a 2-tuple:

$$coverage_l = (\delta_l, g_l) \quad where: \tag{7}$$

- $\delta_l = \langle shape, L \rangle$ defines the area of the location coverage where:
  - $L = \bigcup_{i=0}^{n} l_i \forall i \in \mathbb{N}$ is a set of location stamps
  - $shape$ is a mathematical abstraction used to describe the location coverage, as a continuous coverage area (e.g., rectangle, circle), or non-continuous coverage area (e.g., disk, path, polygon, random)
- g_l is the location granularity or unit of the location coverage (e.g., millimeter, centimeter, meter).

Remark. The shape of a location coverage depends on the sensors and the environment where they are deployed. For instance, the

shape could be lines (for mobile sensor observation tracking), continuous rectangles or squares (e.g., in an office), or non-continuous disks or random shapes (e.g., in a forest excluding lakes). ■

To illustrate the temporal redundancy detection process, consider the CO_2 data collection presented in Table 3. If we apply the temporal redundancy detection algorithm with a deviation threshold $\delta_v = 3$, we detect one temporal redundancy (containing values 109, 110, and 111) and spanning over a temporal coverage of 4 minutes (from 10/02/2019 10:02:00 till 10/02/2019 10:06:00).

a	v	t		l				s
		format	value	format	value			
					x	y	z	
CO_2	98	dd/mm/yyyy hh:mm:ss	10/02/2019 10:00:00	cartesian	8	12	8	S1
CO_2	109	dd/mm/yyyy hh:mm:ss	10/02/2019 10:02:00	cartesian	6	8	6	S1
CO_2	110	dd/mm/yyyy hh:mm:ss	10/02/2019 10:04:00	cartesian	2	4	8	S1
CO_2	111	dd/mm/yyyy hh:mm:ss	10/02/2019 10:06:00	cartesian	4	6	4	S1

Table 3: CO_2 Data Collection

In the following, the redundancy remover summarizes each detected redundancy by a representative data item (tuple) in the output. To illustrate, the (temporally) redundancy-free CO_2 collection is presented in Table 4. In this example, we summarize the redundancy tuples by calculating the averages for each column. The summarising method (e.g., mean, median, centroid) is a system parameter that can be configured in the redundancy cleaner module. Finally, the output can be either stored in a data repository, used to answer a user query, or exchanged with other devices. As a result, the redundancy elimination process applied here has led to a smaller data collection without any loss of useful information. This improves query answering at the device level (cf. Criterion 1), avoids depleting the device resources when answering queries/exchanging data with other devices (cf. Criterion 2), and helps store redundancy-free data in the centralized repositories where advanced services could be applied on the data (cf. Criterion 1).

a	v	t		l				s
		format	value	format	value			
					x	y	z	
CO_2	98	dd/mm/yyyy hh:mm:ss	10/02/2019 10:00:00	cartesian	8	12	8	S1
CO_2	110	dd/mm/yyyy hh:mm:ss	10/02/2019 10:04:00	cartesian	4	6	6	S1

Table 4: Redundancy-free CO_2 Data Collection

4.3 Redundancy Detection Algorithms

The redundancy checker module (cf. Figure 2) consists of two clustering algorithms for the detection of temporal redundancies from static devices, and the detection of spatial-temporal redundancies from mobile devices. The generated clusters contain redundant data based on value similarity. In addition, the temporal and spatial-temporal coverage of each cluster is calculated to keep track of the temporal and/or spatial spread of each redundancy.

Algorithm 1 groups the data into clusters of temporally redundant data. It takes a data collection C as input, and produces a set TR of temporal redundancies (clusters) as output. First, the algorithm sorts all data items in the input collection by ascending time. Then, for each data item, the algorithm checks if a cluster already exists. If not, a new cluster is created with the current data item added as its centroid (lines 3-6). However, if a cluster already exists, the algorithm checks if the current data item belongs to the aforementioned cluster. This is done by measuring the distance between the data item and the cluster centroid values and comparing it to a deviation threshold δ_v (line 8). If the current data item belongs to the cluster, a new centroid is computed and the algorithm checks the next value in the collection (lines 9-10). This step is repeated until the algorithm finds a value that does not belong to the cluster. In this case, the temporal coverage of the cluster is calculated (lines 12-13), the cluster (i.e., temporal redundancy) is added to the output list (line 14), and the variable cluster content (D) is reset (line 15) in order to generate a new cluster and look for other redundancies.

Algorithm 1: Temporal Redundancy Checker

```
Input  : C                    // C is a data item collection (i.e., a set of data items)
Output : TR                   // TR is a set of temporal redundancies found within C
Parameters : δ_v, g_t         // δ_v is a value threshold; g_t is a temporal unit
Local Variables : SC, cov_t, centroid, D, min_t, max_t, δ_t
  /* SC is the temporally sorted collection; cov_t is a temporal coverage; centroid
  is the centroid of a set of values; D is a cluster of data items; min_t is the
  oldest timestamp in a set; max_t is the most recent timestamp in a set; δ_t is a
  temporal interval */
  // Begin algorithm
1 Initialize TR ← ∅
2 SC ← sort_t(C)                          // Sort data items by ascending timestamps
3 foreach data item d_i ∈ SC do
4     if (∄ cluster of redundant data D) then
5         Create new cluster D
6         Initialize centroid ← d_i.v
7     else
8         if (Absolute difference |d_i.v − centroid| ≤ δ_v) then
9             Add data item to cluster D ← d_i
10            Update centroid ← Avg(all d_i.v ∈ D)
11        else
12            Identify temporal interval δ_t ← [min_t, max_t] of D
13            Compute temporal coverage w.r.t. time unit cov_t ← (g_t, δ_t)
14            Add new temporal redundancy TR ← (cov_t, D)
15            Flush out cluster D
16        end
17    end
18 end
19 Return TR
```

Algorithm 2: Spatial-Temporal Redundancy Checker

```
Input  : C                    // C is a data item collection (i.e., a set of data items)
Output : STR                  // STR is a set of spatio-temporal redundancies found within C
Parameters : δ_v, g_t, g_l    // δ_v is a value threshold; g_t is a temporal unit, g_l is a
             location unit
Local Variables : SC, cov_t, cov_l, centroid, D, min_t, max_t, L, δ_t, δ_l, shape
  /* SC is the temporally sorted collection; cov_t is a temporal coverage; cov_l is a
  location coverage; centroid is the centroid of a set of values; D is a cluster of
  data items; min_t is the oldest timestamp in a set; max_t is the most recent
  timestamp in a set; L is a set of locations; δ_t is a temporal interval; : delta_l is a
  location area; shape is a geometrical shape */
  // Begin algorithm
1 Initialize STR ← ∅
2 SC ← sort_t(C)                          // Sort data items by ascending timestamps
3 foreach data item d_i ∈ SC do
4     if (∄ cluster of redundant data D) then
5         Create new cluster D
6         Initialize centroid ← d_i.v
7     else
8         if (Absolute difference |d_i.v − centroid| ≤ δ_v) then
9             Add data item to cluster D ← d_i
10            Update centroid ← Avg(all d_i.v ∈ D)
11            Add data item location to L ← d_i.l
12        else
13            Identify shape ← getShape(L)
14            Identify temporal interval δ_t ← [min_t, max_t] of D
15            Identify location area δ_l ← (shape, L) of D
16            Compute temporal coverage w.r.t. time unit cov_t ← (g_t, δ_t)
17            Compute location coverage w.r.t. location unit cov_l ←
                (g_l, δ_l)
18            Add new spatio − temporal redundancy STR ← (cov_t, cov_l, D)
19            Flush out cluster D
20        end
21    end
22 end
23 Return STR
```

Similarly, Algorithm 2 takes a data collection as input in order to generate a set of clusters as output, where each cluster represents a spatial-temporal redundancy. The clustering principles are the

same in both algorithms. However, the spatial-temporal redundancy checker calculates the spatial coverage for each redundancy (i.e., cluster) in addition to the temporal coverage. This entails keeping track of data location stamps in each cluster (line 11) and calculating the characteristics of the coverage area (lines 13, 15, and 17). Note that both clustering algorithms calculate the temporal and spatial-temporal coverage of each cluster respectively, in order to keep track of the temporal and spatial spread of each redundancy.

5 EXPERIMENTATION & RESULTS

We propose here an experimental protocol in order to evaluate both algorithms. We detail the objectives of the experimentation, the evaluation metrics, and the experiments. Then, we present some preliminary performance results for Algorithm 1.

5.1 Experimental Protocol

5.1.1 Experimentation Objectives. The objectives of the experimentation are two-fold: (i) highlighting the proposal's ability to detect redundancies accurately - this requires evaluating the accuracy of the clustering algorithms when detecting redundancies and comparing them with existing works; and (ii) highlighting the feasibility of implementing the proposed approach at device level - this requires evaluating the performance of our proposal in order to show that the costs are acceptable at the network edge (limited resources).

5.1.2 Evaluation Metrics. In order to evaluate the accuracy of the redundancy detection (i.e., clustering) task we compare the clustering result with ground truth data (from a chosen data set) and measure Precision, Recall, F-measure, and Accuracy accordingly [1]. We adopt these metrics since they are the most commonly used in the literature, and therefore facilitate the comparison with existing works. In order to evaluate the performance of our proposal, we measure the run-time, processor (CPU) consumption, and memory (RAM) consumption.

5.1.3 Proposed Experiments. We propose the following experiments for the evaluation of both algorithms:

- Experiment 1: *Deviation Threshold Impact.* In this test, we evaluate the accuracy of the algorithms by gradually increasing the deviation threshold in order to produce a more compact or relaxed clustering. For each value, we compare the clustering result with the ground truth by measuring the Precision, Recall, and F-measure.
- Experiment 2: *Input Data Size Impact.* In this test, we evaluate the performance of both algorithms by gradually increasing the input data size, and measuring the required time to detect redundancies, as well as the RAM consumption and CPU consumption during each iteration.
- Experiment 3: *Cluster Size Impact.* In this test, we vary the deviation threshold value in order to create clusters of various sizes and spreads. This test allows to measure the performance of the algorithms, and analyze how the creation of more or less clusters affects overall run-time, CPU consumption, and RAM consumption.

5.2 Preliminary Results

We ran Experiment 2 on Algorithm 1 while measuring the execution run-time. The algorithm was developed in Python 3.8 using the PyCharm IDE. We ran the test on a Dell machine running Windows 10 and having a Core i5 8^{th} Generation 1.8 GHZ processor, and 16 GB of RAM. We chose the publicly available Intel Lab Data set [5], and tested the algorithm on 38,656 records. Figure 3 shows the obtained results: where the algorithm's run-time increases linearly with the increase of input data size, such that the required time to detect temporal redundancies remains under 4ms for 38K+ data items. These results are promising and highlight the need for low processing costs at the edge where device resources are limited.

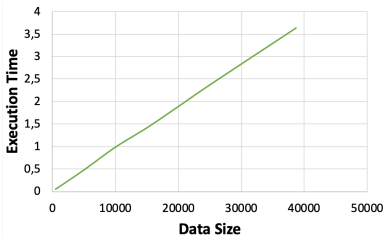

Figure 3: Experiment 2 Results

6 CONCLUSION & FUTURE WORK

In this paper, we address the problem of handling data redundancy in connected environments. We introduce DRMF, a data redundancy management framework which handles sensor data redundancy at the edge device level, considering both static and mobile devices, in order to eliminate redundancies from the source before reaching the core of the network. It includes two clustering algorithms that detect temporal and spatial-temporal data redundancies, and a module for redundancy removal/summarizing. We are currently conducting an extensive experimental study to evaluate our approach. As future work, we plan to investigate the auto-adjustment of the deviation threshold, per device, based on historical runs. In addition, we aim to detect composite redundancies that are generated by data fusion from multiple sensors.

REFERENCES

[1] Enrique Amigó et al. 2009. A comparison of extrinsic clustering evaluation metrics based on formal constraints. *Information retrieval* 12, 4 (2009), 461–486.

[2] S. Chowdhury and A. Benslimane. 2018. Relocating Redundant Sensors in Randomly Deployed Wireless Sensor Networks. In *2018 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 1–6.

[3] Qinlu He, Zhanhuai Li, and Xiao Zhang. 2010. Data deduplication techniques. In *2010 International Conference on Future Information Technology and Management Engineering*, Vol. 1. IEEE, 430–433.

[4] Waleed M Ismael, Mingsheng Gao, Asma A Al-Shargabi, and Ammar Zahary. 2019. An In-Networking Double-Layered Data Reduction for Internet of Things (IoT). *Sensors* 19, 4 (2019), 795.

[5] Farid Lalem and Ahcène Bounceur. 2016. Faulty Data Detection in Wireless Sensor Networks Based on Copula Theory.

[6] Shijing Li, Tian Lan, Bharath Balasubramanian, Moo-Ryong Ra, Hee Won Lee, and Rajesh Panta. 2019. EF-Dedup: Enabling Collaborative Data Deduplication at the Network Edge. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 986–996.

[7] Elio Mansour, Richard Chbeir, and Philippe Arnould. 2019. EQL-CE: An Event Query Language for Connected Environment Management. In *Proceedings of the 15th ACM International Symposium on QoS and Security for Wireless and Mobile Networks*. 43–51.

[8] Prakashgoud Patil and Umakant Kulkarni. 2013. SVM based data redundancy elimination for data aggregation in wireless sensor networks. In *2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, 1309–1316.

[9] Ata Ullah et al. 2019. Secure Healthcare Data Aggregation and Deduplication Scheme for FoG-Orineted IoT. In *2019 IEEE International Conference on Smart Internet of Things (SmartIoT)*. IEEE, 314–319.

[10] Ata Ullah, Iqra Sehr, Muhammad Akbar, and Huansheng Ning. 2018. FoG assisted secure De-duplicated data dissemination in smart healthcare IoT. In *2018 IEEE International Conference on Smart Internet of Things (SmartIoT)*. IEEE, 166–171.