Aggregation of Data Streams in the Web of Things

Felipe Castellanos[1], Mathis Goichon[1], Kévin Kibongui[1], Rivaldo de Souza[1]

[1] National Institute of Applied Sciences of Lyon, Villeurbanne, France
felipe.castellanos-alvarez@insa-lyon.fr
mathis.goichon@insa-lyon.fr
kevin.kibongui@insa-lyon.fr
rivaldo.de-souza@insa-lyon.fr

Abstract. The Web of Things integrates the Internet of Things with Web technologies. Things include sensors and actuators and exchange streams of data. Such data streams are then processed by applications. Most of the time, data streams provide raw data that require to be aggregated to be exploitable by applications. In this PSAT project, we have studied aggregation in the context of the CoSWoT project, where such aggregations of data streams must be computable on devices of small capabilities, ie. with small memory and small processing power.
After a state of the art, we organized our work in an agile process so as to define and implement different ways to aggregate data with techniques that ensure to be efficient in terms of memory.

Keywords: Aggregation, Arduino, C language, Data streams

1 Introduction and Background

1.1 Preamble

The Internet of Things (IoT) is receiving a lot of public and scientific attention today. IoT is a technology that allows us to add a device to an object (e.g. vehicles, factory electronic systems, roofs, lighting, etc.) that can measure environmental parameters, generate associated data and transmit them via a communication network to other systems capable of performing actions (automatic gate, controller, smart radiator). In recent years, IoT applications have multiplied by increasing the number of connected objects in the world. As this trend is not sustainable in the long term with respect to the climate issues of our time, it is important to find ways to drastically increase the efficiency of connected objects while reducing their energy consumption. It is in this context that our project of aggregator will serve as an embedded computer using data received from many sensors.

1.2 Context

The Constrained Semantic Web of Things (CoSWoT[1]) project, funded by the French Agency of Research aims to reduce the energy footprint of the Internet of Things (IoT) by deporting a maximum of data processing to the physical objects necessary for an application to work well (sensors, actuators, gateways) rather than in the cloud.

The CoSWoT project proposes an implementation of constrained servicing, compatible with the imperatives of limiting the consumption of resources on the objects. The data sources (sensors for example) are described in RDF, a model for representing knowledge in graphs.

1.3 Problem definition

The objective of this P-SAT project is to develop an "aggregator", i.e. a module that implements data aggregation algorithms for sensor data streams by managing time windows. For example, this module will have to know how to make an average or other types of aggregation of temperatures over a certain amount of time. The C language has been chosen because this module must be able to run on small equipment, such as Arduino Due or ESP32.
The different requirements we have for this project are:

- make a state of the art of existing aggregation algorithms, and show their specificities for use in constrained environments and on data streams,
- implement a module in charge of data aggregations according to a determined time window. This module subscribes to data streams, aggregates these data, and produces data streams.
- develop generic aggregation algorithms (maximums, minimums, averages minimums, averages...) on time windows.
- test the developed module on data produced in the smart building domain, by the LIRIS or by the Fayol Institute of the Ecole des Mines de St Etienne.
- provide technical documentation allowing this module to be inserted into the architecture of the CoSWoT project, and to easily add new aggregation functions.

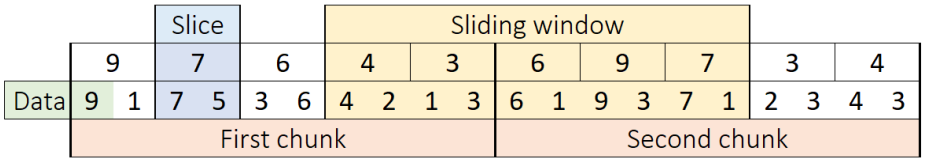
2 State of the Art

2.1 Concepts

This part defines the technical words used during the project to ensure a good understanding of the subject.

[1] https://coswot.gitlab.io/

Fig. 1. Illustration of various state-of-the-art concepts relevant to aggregation algorithms



An aggregation algorithm is a method for continuously computing statistical or mathematical summaries of a rapidly incoming data stream. These algorithms are designed to handle large volumes of data in real-time, without the need to store all of the data in memory.

An aggregation function is a specific mathematical or statistical operation that is applied to a stream of incoming data. The goal of the function is to compute a summary of the data, such as a sum, average, minimum, or maximum value.

A servient is a software component that represents a constrained device or system in the Internet of Things (IoT) network. It acts as an intermediary between the device and the other components of the system. The servient is responsible for managing the communication, data processing, and control logic of the device. The servient is designed to run on resource-constrained devices, such as sensors and actuators, which have limited computing power and memory.

A slice is a subset of a data stream that aggregates raw values using the selected aggregation function. Its length is fixed in a count-value aggregation (i.e in above example, the slice length is 2), or variable in a time-based aggregation (i.e a slice can aggregate from 1 to n raw data depending on what data is sent, it can also have an empty value).

A sliding window is a set of consecutive slices. In the aggregation algorithms, we perform calculations and extract results from the sliding windows data. The length of a sliding window is defined by the number of slices, whenever the aggregation algorithm is instantiated, and it can never change throughout the window's life. When using the time-based strategy, considering that a slice equals for a certain amount of time, a sliding window calculates an aggregation for a slice's amount of time multiplied by the number of slides. For example, if a window includes 24 slices of an 1 hour length, then the sliding window's duration is a day. Also, whenever a new slice is created, it is added to the sliding window using the FIFO strategy; then, the last slide of the window is removed, thus explaining the term "sliding".

A partial aggregation is an aggregation defined over a subset of values that do not represent the entirety of a sliding window. Partial aggregations are, in this context, used in order to produce new slices. Partial aggregations are often used to reduce memory usage or to improve performance.

A chunk is a set of consecutive raw values, however as opposed to sliding windows, the values in the chunk are added but never removed. Chunks are technical concepts used in the SBA algorithm.

2.2 Streaming data: definition, characteristics and aggregation difficulties

Streaming data [9], as the name suggests, is the fact that data is generated continuously from different sources. In a traditional context, streaming data management appears in the realm of Big Data; in which a huge amount of data is collected and processed in real time from many different sources, all at high speed. The collected real time data is then centralized in huge clouds with large computing power to be processed. The problem with this approach is that, although it is efficient and above all financially profitable, it is excessively energy and resource intensive. In our context, the objective is to minimize the use of the cloud by decentralizing as much as possible the data processing directly on the constrained servients, exchanging real-time data continuously between them using a peer-to-peer network.

Windowing [1] is an essential part of data streaming. When it comes to dealing with an unbounded amount of data, we need to set separations between the data to be able to process it efficiently. We are talking of an approach to break the data stream into mini batches to apply different transformations on it. A window opens when the first data item arrives and closes when it meets our criteria for closing a window.
Thus, the great challenge of this project is to successfully implement a solution to process streaming data, in real time, in a constrained environment with low computing power and a very limited memory size.

2.3 Different types of aggregation for streaming data

In this section, we explain two possible classifications for aggregation functions. First, we have invertible aggregations and non-invertible aggregations. Then we have the distributive, algebraic and holistic aggregations. [2]

Invertible aggregations are those for which the merging function of a partial aggregation has an inverse function. For example, for sum, the merging function is addition, and its inverse function is subtraction.

A non-invertible aggregation does not respect this condition. This means that once the merge function of a partial aggregation is performed, it is impossible to perform an inverse operation. As an example we can consider the median.

An aggregation is distributive if it would have the same result if we first split our data in multiple partitions, aggregate them and do a global aggregation with the results obtained, or if we just do an aggregation of the entire dataset. To explain this with an example, in sum aggregation, we can split our dataset in multiple parts and then the result of aggregating the sum of all the subparts will be the same as the sum of all the dataset.

Algebraic aggregations have a bounded size for partial aggregations (e.g. mean, geometric mean). An aggregation is considered algebraic if it can be calculated by an algebraic function with N arguments (with N a positive integer), each of them obtained by applying a distributive aggregate function. The mean illustrates this type of aggregation. If we split our dataset in N equal parts with the same amount of compo-

nents, we would have the same result if we calculate the mean of the whole dataset or if we calculate the mean between the mean of every subpart of the dataset.

Holistic aggregations have an unbounded size for partial aggregations (ex: median, quartile, percentile). We can consider an aggregation holistic if a sub-aggregate has not a constant size, in other words we don't have an algebraic function with N constant arguments that characterizes the computation.

In **Table 1**, classical aggregation functions are classified according to these classes.

2.4 Algorithms for aggregation of streaming data

To build the aggregation functions defined above, we need to define which algorithms we are going to implement. Thus, we made a state-of-the-art review of the existing algorithms, namely PBA, SBA, SlickDeque, TwoStacks.

The Parallel Boundary Aggregator (denoted PBA) algorithm [6] was developed to work on non-invertible aggregations that are either distributive or algebraic. In **Table 1** of the classical aggregations we can see that this concerns the min and max functions. In PBA, a stream is considered as a sequence of chunks having an identical number of slices. In PBA we assume that the system is equipped with at least two cores allowing to run two tasks in parallel. This is extremely important for the algorithms to work. PBA is a constant time solution, which is achieved by maintaining two buffers: csa (cumulative slice aggregations) and lcs (left cumulative slice aggregations). Generally, a csa buffer is computed through accumulating slice aggregations from left to right inside a chunk (an array of slices), and a lcs buffer from right to left. Also it appears that with modifications, the PBA algorithm could be applied to invertible functions, such as Mean for example, but more efficient algorithms might be applied for these functions. Finally, PBA does not apply to non-invertible and holistic functions.

The Sequential Boundary Aggregator (denoted SBA) algorithm is the non-parallel version of PBA, which therefore uses a single thread to calculate the aggregations. SBA follows the same procedure as PBA, except that the task of computing lcs in SBA is done in the same thread as the one that computes csa buffer. In the Implementation part of this paper we are going to explain our implementation of SBA.

The SlickDeque (Inv) algorithm [2] seems to be the most efficient state of the art algorithm on all types of invertible aggregations. In **Table 1** we can find the different classical aggregations concerned. SlickDeque builds a shared execution plan, it includes a full list of partials augmented with their lengths and lists of queries to be evaluated for each partial. After that, it generates a data structure initialization with a circular array, partials and a map. The arriving partials aggregates will be inserted into the partials array. The execution consists in a loop that continuously returns all query results while they are expected. It loops over all ranges to answer mappings in the answers map. SlickDeque (Inv) only works for the invertible queries, as it uses the aggregate and an inverse operation, however a Non-Inv version of SlickDeque is also presented in the same paper, which uses the same principles as SlickDeque (Inv), and is designed to work on non-invertible aggregations.

TwoStacks [9] is another state-of-the art algorithm, whose principle is to divide the whole sliding window into two stacks with different roles: one that tracks the earliest part of the window, and one that tracks the latest part of the window. The result of the aggregation takes in account every element from both sliding windows. Whenever data is added, it is pushed onto the top of the latest stack. Then the algorithm checks the top of the earliest stack; if its timestamp leaves the sliding window, then it is removed from that stack.

The FlatFIT [3] algorithm also divides its sliding window into smaller sub-windows, which are named buckets. Each aggregation is calculated on each bucket separately. This is based on the principle that there should be no need to update the aggregation on the entire window, which could be costly, so we would rather update one bucket at a time. FlatFIT is very efficient with its memory management as it does not use a significant amount of memory.

Furthermore, none of the algorithms presented above is able to aggregate holistic functions (median, quartiles, top-k). The use cases of the CoSWoT project that were formulated did not require the support of this type of aggregation, so we have chosen not to investigate this category.

3 Retained approaches

For non-holistic non-invertible algorithms, we had multiple options. The best ones were SlickDeque (Non-Inv), PBA/SBA and TwoStack. To choose between those, we read through the different benchmarks in the state-of-the-art papers [6]. Those benchmarks allowed us to conclude that PBA/SBA are the best for time and space complexity. Unlike SlickDeque for example, PBA/SBA runs in constant time; such a feature of PBA/SBA guarantees its performance in the worst case. Note that amortized and worst-case time determine throughput and latency, respectively. Tests executed [6] over the DEBS'12 dataset conclude that SBA, the non-parallel version of PBA, is the fastest as long as the window size is inferior to 2^{11}. If it goes beyond 2^{11}, then PBA becomes the fastest algorithm. In the current use cases of the project, the volume of data to be aggregated is relatively small. For time-based, the needs expressed do not require a large window size. For count-based, the needs expressed also require small windows. Moreover, we work on constrained servients which rarely have several cores. Thus, for all these reasons, we have chosen to implement SBA[2].

For non-holistic invertible aggregations, our options were SlickDeque (Inv), TwoStacks and FlatFIT. We also referred to the state-of-the-art benchmarks [2], which show that SlickDeque (Inv) is slightly faster than FlatFIT and TwoStacks which both present very similar results. From a memory point of view, referring to this paper [2], the non-inv SlickDeque algorithm is at least as efficient as FlatFIT and TwoStacks. Indeed, a linear memory size of *2n* is necessary for these two algorithms, whereas SlickDeque needs *2n* in the worst case, and is in constant size (*2*) in the best case. Moreover, the probability of the worst case is negligible (*1* in *n!*). We have

[2] It should be noted that the technical gaps between SBA and PBA are small and that it is possible to switch from one to the other if the needs require it.

therefore chosen to retain SlickDeque as the aggregation algorithm for non-holistic invertible functions.

After analysis of the different aggregation functions and implementation algorithms, here is a summary of the chosen algorithms for each aggregation function.

Table 1. List of selected aggregations with the associated algorithms.

Aggregation Name	Inversible	Type	Algorithm
Sum	Yes	Distributive	SlickDeque
Count	Yes	Distributive	SlickDeque
Product	Yes	Distributive	SlickDeque
Sum of Squares	Yes	Distributive	SlickDeque
Average	Yes	Algebraic	SlickDeque
Geometric Mean	Yes	Algebraic	SlickDeque
Standard Deviation	Yes	Algebraic	SlickDeque
Maximum	No	Distributive	PBA
Minimum	No	Distributive	PBA
Range	No	Algebraic	Not implemented
Median	No	Holistic	Not implemented
Quartiles	No	Holistic	Not implemented
Top-K	No	Holistic	Not implemented

4 Implementation

This part describes the implementation choices made after the theoretical study performed previously. The implementation of our module is now denoted CoSA (Constrained Semantic Aggregator). It is important to note that we have implemented our solution using dynamic allocation.

4.1 Software architecture

As a global view, our implementation is divided into four parts:

- aggregation functions, which define how the data should be aggregated;
- the aggregation algorithms, as described above, deal with sliding windows, in count based or time-based modes, by aggregating the data, relying on the aggregation functions, and producing aggregated data for each window;
- the manager, which, as the name suggests, manages the aggregations, transmits the received data to the right aggregation algorithms and retrieves the aggregated data when they are available;
- the main function, or calling application, which is in charge of initializing the necessary aggregations, receiving the data from the sensors to aggregate them and producing the aggregated results.

To have a better understanding of the global functioning and the different interactions between the four parts, we have made a simplified sequence diagram (**Fig. 2** and **Fig. 3**). The implementation details will be presented in more detail in the following section. For clarity we have encapsulated the aggregation algorithms (SBA and SlickDeque) under the name *Aggregation_Algorithm*. The same applies to the aggregation functions (Min, Max, Average, Count, ...) which have been encapsulated under the name *Aggregation_Function*. In our implementation we have also implemented this abstraction to guarantee genericity. About the *Calling_Application*, we conceived our solution as an independent module and therefore it can be anything (main function, global servient, ...). The Integration on CoSWoT Project part details more about this point.

Fig. 2. Simplified sequence diagram of our implementation - Initialization part

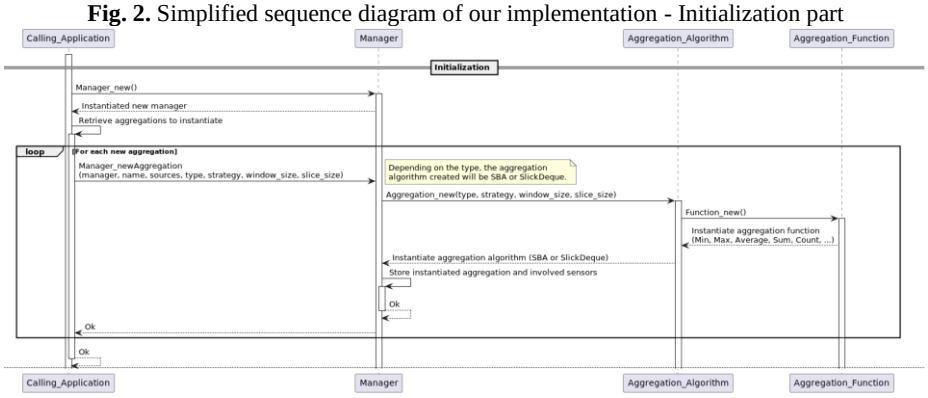


Fig. 2 shows the sequence for initializing our solution. Its objective is to create a set of aggregations that will have to be performed over time. The calling application must retrieve the various aggregations to be performed in order to add them to the *Manager*. In our implementation we read a *.json* file, but in the global context of the project this configuration should be retrieved by communicating with RDF triples. The *Manager* takes care of instantiating the aggregations (SBA or SlickDeque) according to the data it receives. It will then store the different aggregations and, for each known sensor, the list of aggregations to which it belongs. The implementation details will be described in the following section.

Fig. 3. Simplified sequence diagram of our implementation - Runtime part

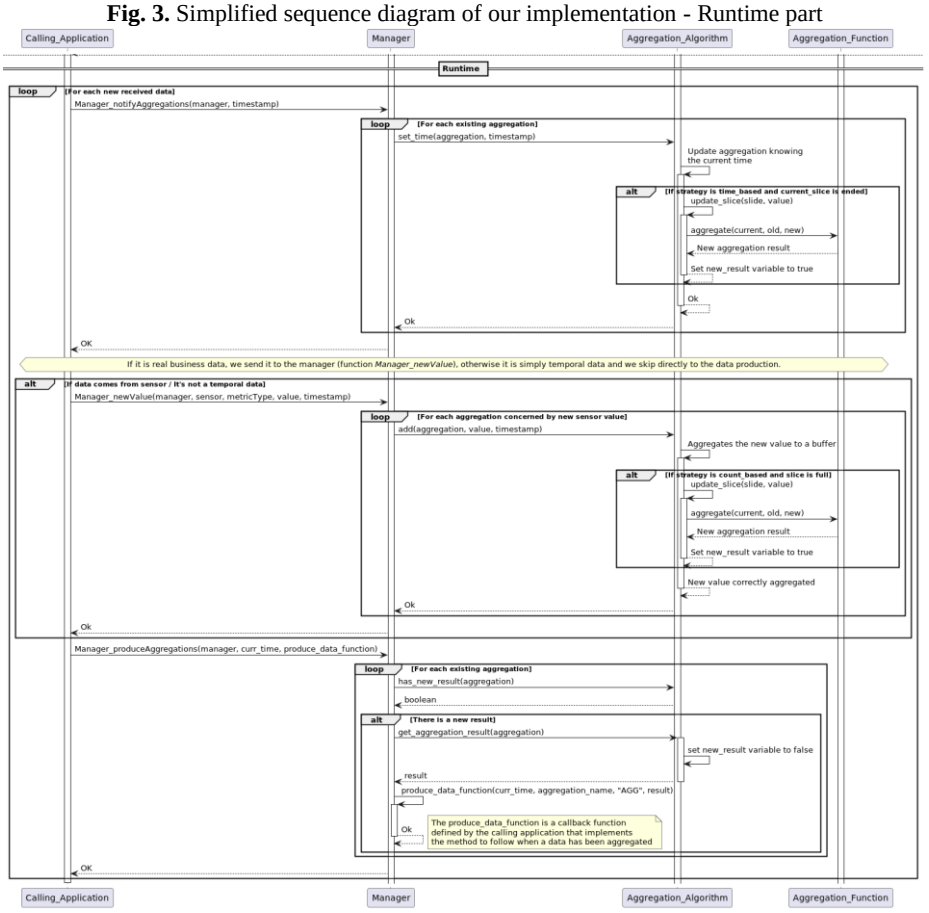


Fig. 3 shows the sequence for the runtime, logically of infinite duration, of our solution. The objective here is to process each new data received, to check if the processing should result in the production of a new aggregated value, and if so to produce this value.

To be clearer, we distinguish two types of data: business data and strictly temporal data. The business data are the data that are intended to be aggregated, they are produced by sensors or other servient. Strictly temporal data are data received at regular intervals that allow us to see the passage of time, like a heartbeat.

When a new data arrives (business or strictly temporal), we start by notifying all the aggregations of the time it is. The time-based aggregations can then update their window, especially if the time of the last slice is out of date. Then, if the received data is a business data from a sensor, it is added to the set of aggregations in which the sensor is implied. Count-based aggregations can then update their windows, especially if the current slice is full and a new one is created. Finally, we check each aggregation to ask them if they have a new aggregate result to produce; and if so we give them a function to call. This function is defined by the *Calling_Application* and al-

lows defining a different data production method depending on the context in which the module is used.

In our main application, we have chosen to read the input data into a *.csv* file, and the data is produced and written into another *.csv* file. This allowed us to test our application (more details in the Tests section). In the final context of the CoSWoT project, it will be needed to read data from RDF graphs, and produce new RDF graphs. Our implementation is flexible enough to allow this.

4.2 Algorithms implementation

This part describes in detail the implementation choices we made on the main elements of our solution (Manager, SBA, SlickDeque).

Manager implementation

As seen in the previous section, the manager offers four services (excluding constructor and destructor). To manage the different aggregations we used two attributes: an aggregation collection (denoted aggregations) and a collection of sensor / aggregations pairs (denoted sensors). For the aggregations attribute, we used a HashMap with the name of the aggregation as a key (unique value by business design) and a pointer to the instantiated aggregation as a value. For the sensors attribute, we also used a HashMap with as a key the name of the sensor (unique value by business design) and as a value a LinkedList of names of aggregations in which the sensor is implied. Thus, when a new data from a sensor is received, it is only necessary to look at the aggregations concerned by this data, and to update each of them.

Since the language offers only a few structures in the standard libraries, the HashTable and LinkedList codes have been implemented by ourselves, using various online resources. We are fully aware that these choices are not the most relevant in a constrained environment, especially for HashTables which, although efficient, waste space unnecessarily. However, our realization was above all about the implementation of aggregation algorithms and we have therefore privileged a more standard and efficient approach than the context would impose.

Finally, about the *Manager_notifyAggregations* function, some deepening is necessary. This function allows, among other things, to notify the aggregations of the current time, so that they can update their sliding windows. However, if no data is received during a complete slice (or even several), this would create gaps and there would be windows that would not be produced. For example, if a data is received and that one skips 3 slices, there would be 3 complete windows which would not be produced and then lost. Thus, we have assumed that the manager receives at least one value (business or temporal) per slice. This is a necessary condition for the proper functioning of the module. This is why we made a distinction between business data and temporal data, and why we used the image of a heartbeat.

Aggregator implementation
The Aggregator header file has been written to provide a structure for the aggregation algorithms, so that the calling scripts can use every aggregation algorithm in the exact same way, allowing genericity in the code. The services that must be provided are the following ones:

- A pointer to a function named *add*, which must add a new value to the window, and eventually add a new slice if the required conditions are met. One function pointer weighs 8 bytes.
- A pointer to a function named *get_aggregation_result*, which returns the result of the current aggregation.
- An integer named *new_result*, which tracks if a new slice has been created and has not been sent to the calling script. It is set to 1 whenever a slice is created, and is set to 0 whenever the calling script uses *get_aggregation_result* to retrieve the most recent result. One integer weighs 4 bytes.
- A pointer to a function named *has_new_result*, which returns the content of *new_result*. The calling script can use this function to decide whether it calls *get_aggregation_result* or not.
- An integer named *time_based*, which tracks the algorithm's aggregation strategy. This variable is only set at the construction of the object, and never changes. If set to 1, the aggregation strategy is time-based, otherwise it is set to 0, and the strategy is count-based.
- An integer named *timestamp*, which tracks the current time in an integer. While executing in time-based mode, the algorithm must be aware of the current time, in order to decide when it can create slices.
- A pointer to a function named *set_time*, which allows the calling script to tell the current time to the aggregator. This function must take a timestamp as a parameter. The algorithm will then set its timestamp to the new value.
- A pointer to a function named *view*, which displays in the console the aggregator's current state.
- A pointer to a function named *free*, which must provide a way to completely destroy the aggregator's instance, and frees every memory it has used.

As the function pointers weigh 8 bytes and the integer variables weigh 4 bytes, the sum of each element of the structure reaches 60 bytes. As 4 additional bytes are added due to structure padding [7], the total size of the Aggregator structure is 64 bytes.

SBA implementation
As it is an aggregation algorithm, the SBA implementation uses the Aggregator structure, and therefore implements each of its attributes and pointers. However, it adds some new attributes in order to function properly:

- An integer named *nbSlices* which memorizes the amount of slices in each window.
- Two integers named *r* and *s* which respectfully memorize the number of elements in the window, and the number of elements in one slice. In time-based, *r* memoriz-

es the total length of the window in seconds, and *s* tracks the total length of one slice in seconds.

- A double-ended queue pointer named *c*, which tracks the raw values that were not added in a slice yet.
- A pointer to an array of double-ended queues named *saa*. Every double-ended queue in *saa* represents a chunk, and aggregates slices from *c*. A maximum of three chunks can be memorized at once.
- Depending on various conditions, *saa* can either remember the values of the slices, or the cumulative aggregated values from right to left [6]. In an unrestricted environment, each *saa* double-ended queue should've been divided into two different arrays, which would be *saa* (slice aggregation array) and *lcs* (left cumulative slice aggregations). However, those two arrays are actually never used at the same time, which is why we can use one memory space for both. This is a 4 (size of a float) * 3 (maximum remembered chunks) * *nbSlices* memory save in bytes, which can definitely be very worthy if the window size is big. Though sometimes, code optimisation can be at the expense of code clarity, which is the case for this specific variable. This is part of the reason why we made sure to write a lot of documentation so the code stays clear.
- A float value named *csa*, which represents the right cumulative slice aggregation's current value. In an unrestricted environment, this would also be an array, which saves 4 * (*nbSlices* - 1) bytes of memory.
- An aggregation pointer named *aggreg* which tracks the current aggregation function that is used by the current SBA instance. It weighs 24 bytes of memory.

As such, the total SBA structure weighs 128 bytes by itself. It also allocates 24 bytes of memory for the aggregation algorithm, 96 bytes for the *c* double-ended queue and the *saa* variable which both allocate 24 bytes of memory per element.

The biggest stakes we have while developing an aggregation algorithm is to match as closely as possible the theoretical results in the paper in terms of both space and time complexity. While implementing though, we always encounter specific constraints that are only relevant in the CoSWoT context, and we have to manage those without reducing the program performance. In the paper [6], SBA is defined to theoretically require a total space of $\frac{(3n+13)}{2}$, *n* being the number of slices. We are close to this result practically. Removing the *c* double-ended is the biggest improvement we could make in order to lessen the SBA memory usage.

Some other changes could be brought to the SBA implementation; for instance the number of slices is not necessary to be kept in memory as we could recalculate it with *r* and *s*, but that also uses computational power, so a choice has to be made.

SlickDeque implementation

The SlickDeque implementation also uses the Aggregator structure, to which it adds additional attributes:

- A pointer to a *Slick_Aggregation* named *aggreg*, which similarly as SBA's *aggreg*, is the aggregation function used by the current SlickDeque instance. That aggrega-

tion must be able to aggregate and return a new result using the current partial result, the old value that is getting replaced, and the new value. It must also provide a function that returns an initial value for the aggregation, for example Product's initial value is 1.

- A pointer to a double-ended queue named *buffer* which contains values that have not yet been aggregated in a slice. In the future, this variable should certainly be removed, as we could aggregate values in a partial slice instead of memorizing them.
- An array named *partials* which tracks every slice.
- An integer named *width* which tracks the length of the window.
- A float named *result* which tracks the last calculated result of the aggregation. This result is then sent on the *get_aggregation_result* algorithm.
- An integer named *currentPos* which tracks the current position of the next value to change in the window.

Note that the implemented *Slick_Aggregations* are either distributive or algebraic. If they are algebraic, they by themselves include other distributive *Slick_Aggregations* in their implementation; for example, the Average aggregation allocates memory for a *Sum* and a *Count* aggregation.

The paper defines the memory complexity of the SlickDeque algorithm, which does not exceed *2n* in the worst case, n being the number of slices. We only need to memorize one dynamic array, named partials, in order to track every slice, unlike SBA where we have multiple double-ended queues in order to store the necessary data. On this algorithm, we also got close to the actual memory space theorized in the papers. Similarly as SBA, the improvement we can make is to remove the double-ended queue *buffer* and progressively aggregate data before creating a slice.

4.3 Integration in the CoSWoT project

The aggregator has been developed as a library and can therefore be easily integrated into any environment. Thus, during the development and testing phase, this project was stand-alone, which allowed progress without being impacted by the rest of the project.

In order to be integrated in the CoSWoT project, the services proposed by the manager must be used. In this way, it is necessary to subscribe to the different servients in order to receive the initialization data, the data collected by the sensors, and then produce the aggregated data so that other servients can exploit them. In contrast to the standalone mode, on the global project, data comes from other constrained servients. The communication is done through RDF [8] graphs (input and output), and it is necessary to subscribe to the different sensors, using callback functions.

Currently, the integration with the CoSWoT project remains delicate as the communication protocols and data formats are not yet clearly defined. Indeed, for the moment the JSON-LD language is used but it should soon give way to CBOR-LD which is more compact. Similarly, the format and production of temporal data is still under discussion. The aggregator has been integrated as a sub-module of the CoSWoT

project and everything is in place for a quick integration. As soon as the above mentioned issues are resolved, the integration will be almost immediate. A partial integration has been done to illustrate how to use the aggregator. The current blocking points are more business related and concern the interaction between the different modules.

5 Experimental evaluation

This part describes the test procedures we have implemented to evaluate our solution.

5.1 Tests

We consider functional testing to be an essential part of the development of CoSA. As the aggregator must provide very precise results and never make mistakes, we have to ensure that every test case is validated at all times. This serves as a proof that CoSA is working properly, which reinforces confidence in our work. Testing also helps us during development, to locate the issues we encounter. Thus, we have written numerous tests over our aggregation algorithms. We had to find a way to execute proper tests in C with a lightweight framework. For this use we chose Unity [10], which is easy to install as it's only composed of three files (two headers and one source), very lightweight, simple to use, and includes features for embedded development.

We first tested both of our algorithms, SBA and SlickDeque, over multiple test scenarios, using both count-based and time-based aggregation strategies, over various edge cases. The tests also allow us to track the memory usage of our algorithms using memory analysis tools. Then, we wrote acceptance tests for the Manager part, which first retrieves the initialization files and instantiates the aggregator instances, then receives data from the sensors, sends them to the aggregations, and finally sends the data back to the servient. We send to the manager an initialization file in *.json* and a *.csv* file containing all the data sent from the servient, then the manager outputs a *.csv* file with its responses, and we check if that *.csv* file matches with another *.csv* file containing the expected responses that we have generated ourselves manually. This is a testing method that ensures that the program returns exactly what we expect it to, and also allows us to visually check the generated results by comparing the *.csv* files manually.

5.2 ESP-32

In order to evaluate the aggregation algorithms we have implemented, we have adapted the project to an arduino target. More precisely, on an ESP32 development module which in addition to offering basic functionalities such as mathematical calculation, reading/writing offers Wi-Fi and Bluetooth connectivity (which makes it a connected object). In carrying out this integration, we encountered a number of problems.

Firstly, since our test benchmark is designed to support input and output files, we had to add a plugin to our arduino IDE to be able to read and write to the files and run

tests (ESP32 Filesystem Uploader). Secondly, since the Arduino compiler supports both C and C++, we had to add directives in all our header files explaining that we have C code in our project to ensure compilation on both the computer and the arduino. Thirdly, some constraints of the arduino forced us to adopt some good practices. For example: when passing a variable to a function as a string, it is ideal to use a constant to make it immutable.

The key library for our tests was *SPIFFS*. *SPIFFS*, for Serial Peripheral Interface Flash File System, is a light file system adapted (among others) to microcontrollers with SPI flash memory such as the ESP32 and ESP8266. The Arduino *SPIFFS.h* library allows to access the flash memory as if it was a normal file system like the one of a computer (but much simpler of course). We can read, write and add data to a file and perform some simple operations (format, rename, retrieve information...). Another important aspect of our integration in esp32 is that in production, it is possible to change the data source via Wi-Fi or Bluetooth.

In order to facilitate the future use of our project in Arduino, we have added some essential files such as:

- *examples/aggregator/aggregator.ino* to have an Arduino example sketches;
- *library.properties* to configure the project as an Arduino library;
- *keywords.txt* to list specific keywords and other global constants if needed;
- *README.md* to guide future users, especially to run the tests;
- *AUTHORS* to cite the authors.

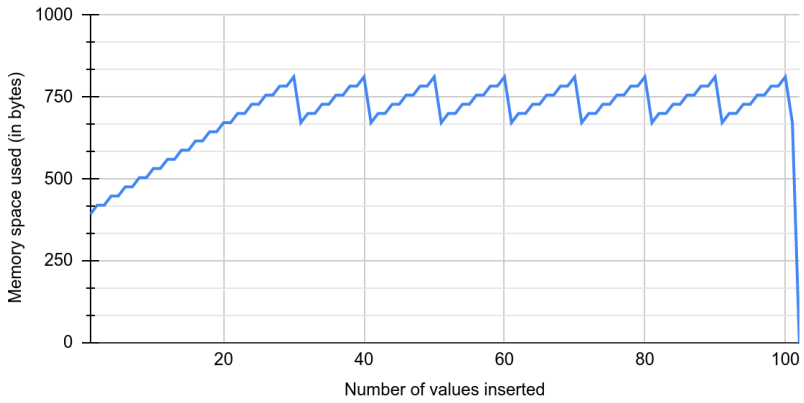
5.3 Memory benchmarks

The memory footprint of the SBA and SlickDeque algorithm implementation is described here. The calculation concerns only the functioning of the algorithms themselves and does not consider the implementation details engendered by the rest of the project. The tests were performed directly on an ESP32.

To evaluate the performance of the aggregator, we performed a benchmark focused on the time cost and the memory cost. We instantiated one or more SBAs to verify that the memory space occupied corresponds to our theoretical calculations. Then we checked that the manager in charge of scheduling and managing several instances (SBA and SlickDeque) worked correctly even in quasi extreme cases. We artificially increased the input data to see if we could reach the limits of our ESP32 with a too loaded instance or if it is only once we reach a high number of instances that the problem arises. Since the values processed are real, the algorithm also works for integers. The float values in c range from $\pm 1{,}8 * 10^{-38}$ to $\pm 3{,}4 * 10^{38}$. Our tests are based on values that do not exceed this range.

In the first step we performed a test with 100 values. The initial parameters are: a count-based aggregation, a window size of 10 values and a slice size of 2 values. The objective of this test is to see the evolution of the memory on a standard case. We expect to observe a maximum value of occupied memory size.

Fig. 4. Evolution of memory consumption - SBA Count-based 100 Values

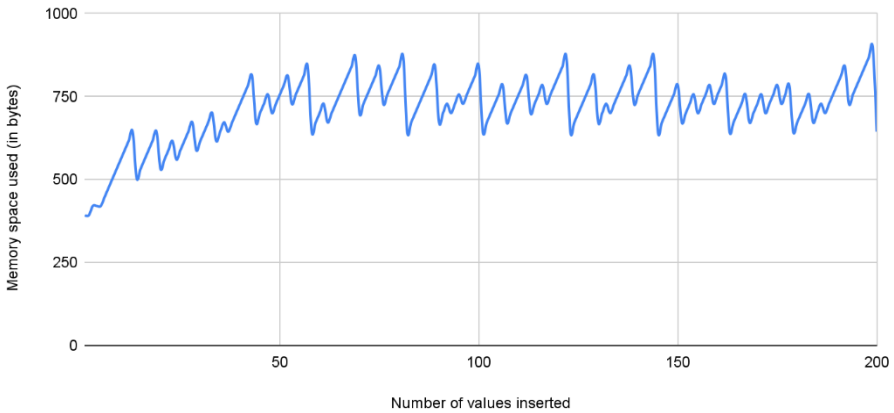


We can see on this graph that SBA occupies 392 bytes of memory when it is empty. In the early stage from 0 to 30 added values, we dynamically allocate memory bytes to memorize the necessary values for SBA to work, which mainly implies filling *saa* with slide values.

When we reach the 30th added value, SBA reaches its maximum size, at which it will plateau. This maximum is reached on the 30th value because the window has a size of 10 values, and SBA must memorize 3 chunks in order to function properly. SBA is thus filled at the 30th value. From there, it frees space on the *saa* variable, which explains the sudden drop in memory usage. Then a new chunk is filled at the 40th value, and *saa* is freed again, and this cycle repeats until the program execution stops, at which point every byte used by SBA is freed.

Then, we tested the SBA algorithm in time-based with 200 values. We have between 0 to 5 values per slice. We expect to see a curve similar to **Fig. 4** with an increase to a plateau value and then variations for each new slice.

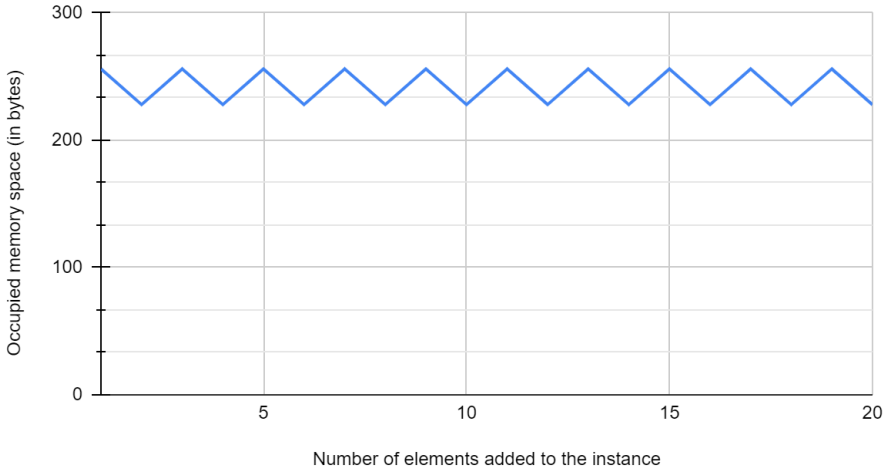
Fig. 5. Evolution of memory consumption - SBA Time-based 200 Values



We can see in **Fig. 5** that we do have an increase at the beginning of the graph, however we do not get the expected plateau value. This can be explained by the fact that we missed an implementation detail. Indeed, when a new temporal data arrives, it is stored and aggregated only at the end of the slice. However, the expected behavior is that the data should be aggregated directly in a partial way in the current slice. This is a minor issue to be fixed and will be implemented before the project is released.

After that, we realized a benchmark on the SlickDeque in order to determine its memory usage. Our goal is to evaluate whether the SlickDeque algorithm memory curve acts the same as the SBA ones, and to ensure that the memory usage of SlickDeque does plateau as planned.

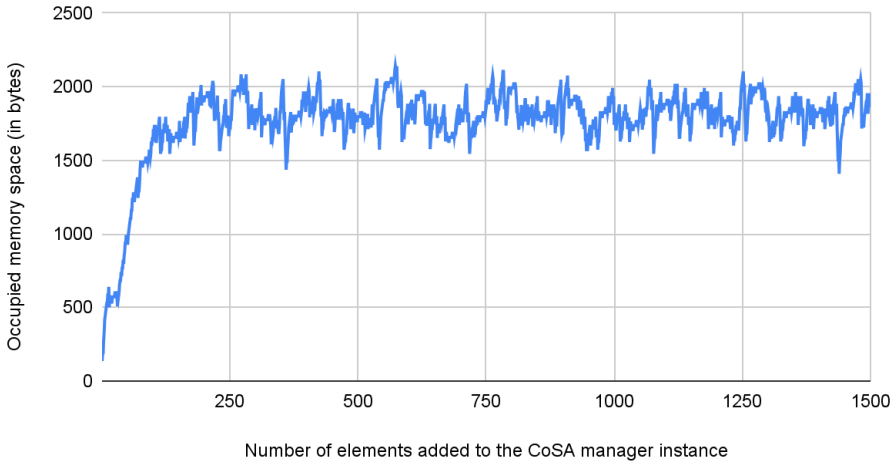
Fig. 6. Evolution of memory consumption - SlickDeque Count-based 20 Values



As we planned, the memory usage of SlickDeque does plateau. Unlike PBA which takes time to allocate the memory that it will end up using, SlickDeque immediately is initialized at the value to which it will plateau, which is 265 bytes. SlickDeque can never exceed this value, no matter how many values we add to it, similarly to PBA. The only memory fluctuation that we can observe is related to SlickDeque's usage of a double-ended queue *buffer*, which allocates memory when values are added, and frees memory when slices are created.

At last, we decided to make a final benchmark that encapsulates the whole CoSA program. This benchmark will show the memory usage of a manager instance in action, initialized with 10 aggregators with different window lengths, slices and types, and 17 different data sources. This benchmark aims to reproduce as closely as possible a real use case of CoSA.

Fig. 7. Evolution of memory consumption - CoSA manager instance - 1500 Values



We can observe that the manager instance seems to plateau around 2000 bytes, which shows how lightweight CoSA is. The memory that the CoSA uses is perfectly fine in order to run it on an ESP32 with several additional modules. No matter how many more values we add, the manager will never use more bytes than shown here.

6 Conclusion

6.1 Difficulties and evolution perspectives

Working in a constrained context is not something simple. Indeed, it imposes a rather particular way of developing which strongly disrupts the classical programming paradigms. Thus, a refactoring work would be necessary to get closer to an implementation that better respects the integrated standards. We are thinking in particular of our choice to have implemented our solution based on dynamic allocation. Knowing that we can determine in advance the size needed to perform an aggregation according to its parameters, a static allocation would be relevant. We also believe, in view of our implementation, that the technical gap to achieve this is rather small.

Furthermore, our current implementation does not support latency. Indeed, let's imagine a temporal aggregation with a window ending at a time *t*. If a data is produced at *t-1* by a sensor but arrives only at *t+1* to be aggregated, it will not be considered in the window dated at *t*. This data, which took some time to be received, will then be lost. To overcome this problem, a certain latency could be implemented before producing an aggregated data, allowing the late data to be aggregated in the right window, and thus reducing the lost data. For example, a window ending at time *t* could be produced only at time *t+2*.

Besides, the support of holistic aggregations is a relevant evolution. This would in particular allow to support the median which is an aggregation function that can be

useful. It could have been possible to run SBA with a histogram to implement the median and get an approximate value. However, in order to get a credible long term solution, it is necessary to study the literature in order to identify and implement state of the art algorithms that are efficient for this type of function.

Finally, to push the use cases even further, it would have been very interesting to perform aggregations of aggregations. For example, it would be possible to collect the minimum temperature on each floor of a building, and then to perform an average on these minimums. Or, on the same servient, collect the average of the ambient humidity every hour, and produce a maximum of this average level once a day. Currently, this need should be covered by the current implementation. Unfortunately, no test could be performed due to lack of time, so it is impossible to say that it works.

6.2 Project management and results analysis

This project was managed in different ways. For group communication (internal and with the managers) we used the Slack platform. In addition to that we had one meeting per week with the project managers and another internal meeting. Those meetings were technical or organizational. With both meetings we were able to handle the different tasks of the project and issues found. We used GitLab to organize the different tasks and issues found, handle the different versions of our code and document it. As our work is part of the larger context of the CoSWoT project, we also had to communicate with the other PSAT group working on the integration of the different modules.

Answering the question proposed at the beginning of this paper, we can conclude that it is possible to reduce the energy footprint of IoT by processing the maximum amount of data to the physical objects. To accomplish that, we need to aggregate data in a memory efficient way because of the hardware limitations of those devices. The way found to do that is by using the aggregations types described in this paper.

Through this project, we learned a new way of working with data. This is extremely interesting because nowadays, big data is becoming increasingly important, with more and more computing power, resources and cloud. This project allowed us to work in an opposite way. With the current ecological crisis, thinking about constrained environments becomes particularly meaningful.

Acknowledgements

We want to thank the CoSWoT ANR project for having provided the topic of this work and data to experiment our work. We also thank the other PSAT team on CoSWoT frugal IoT and their tutor Lionel Médini for the collaborative work we made together. Moreover, we thank Ghislain Atemezing and Alexandre Bento for their help throughout the project. And finally, we deeply thank Yann Gripay and Frédérique Laforest for their tutoring and support during all the project.

References

1. Asif, M.H.: Windowing in Flink, https://medium.com/big-data-processing/windowing-in-flink-8896e0fed787, last accessed 2023/01/23.
2. Shein, A. et al.: SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation, https://openproceedings.org/2018/conf/edbt/paper-197.pdf, (2018). https://doi.org/10.5441/002/EDBT.2018.35.
3. Shein, A.U. et al.: FlatFIT: Accelerated Incremental Sliding-Window Aggregation For Real-Time Analytics. In: Proceedings of the 29th International Conference on Scientific and Statistical Database Management. pp. 1–12 Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3085504.3085509.
4. Tangwongsan, K. et al.: Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time. In: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems. pp. 66–77 ACM, Barcelona Spain (2017). https://doi.org/10.1145/3093742.3093925.
5. wojciech-marusarz: Challenges of Data Stream Processing: Big Data Streams 1:1, https://nexocode.com/blog/posts/data-stream-processing-challenges/, last accessed 2023/01/23.
6. Zhang, C. et al.: Efficient Incremental Computation of Aggregations over Sliding Windows. In: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining. pp. 2136–2144 ACM, Virtual Event Singapore (2021). https://doi.org/10.1145/3447548.3467360.
7. Data structure alignment, https://en.wikipedia.org/w/index.php?title=Data_structure_alignment, (2022).
8. RDF - Semantic Web Standards, https://www.w3.org/RDF/, last accessed 2023/01/23.
9. Streaming data, https://en.wikipedia.org/w/index.php?title=Streaming_data, (2022).
10. Unity, http://www.throwtheswitch.org/unity, last accessed 2023/01/23.