Historical Data Storage for Large Scale Sensor Networks

Loïc Petit

LIG - SIGMA
Grenoble University
220 rue de la Chimie
38400 Saint Martin d'Hères,
France
loic.petit@imag.fr

Abdelhamid Nafaa

Computer Science and
Informatics
University College Dublin
Belfield 4
Dublin, Ireland
nafaa@ieee.org

Raja Jurdak

CSIRO ICT Center
QCAT Technology Court
Pullenvale QLD 4069, Australia
rjurdak@ieee.org

RESUME

Les réseaux de capteurs sans fils sont de plus en plus déployées pour de nouvelles applications comme la foresterie ou l'agriculture de précision, répondant à différents besoins en termes de surveillance et collecte de données à distance. Bien que plusieurs travaux de recherche ont couvert le routage de données dans des réseaux de capteurs à grande échelle, peu de travaux se sont intéressés aux architectures de stockage sous-jacentes qui souvent constituent le goulot d'étranglement du fait des accès disques intensifs. Cet article décrit une architecture de stockage et un accès aux mesures émanant d'un réseau de capteurs sans fils à très grande échelle. Notre contribution s'articule autour (i) d'un partitionnement des données couplées a une méthode originale basée sur un double-buffer pour réduire le coût en puissance de traitement et réduire les délais de bout-en-bout et (ii) d'un schéma optimisé pour les requêtes et l'accès rapide aux mesures sauvegardées. Nous avons implémenté notre système de sauvegarde, ainsi que d'autres mécanismes de routage au niveau réseau des capteurs, dans un prototype disponible au campus d'UCD. Les résultats d'évaluation de performances ont été effectués à l'aide d'un émulateur de réseaux de capteurs.

MOTS CLES : capteur, stockage historique, partitionnement, double-buffer

ABSTRACT

Wireless sensor networks are rapidly finding their way through a plethora of new applications like precision farming and forestry, with increasing network scale, system complexity, and data rate. While scalable MAC and routing protocols for sensor networks have been well addressed in recent years, the scalability of the back-end storage architecture has been largely overlooked. As a result, current storage and retrieval architectures usually lead to an excessive I/O cost when it comes to improving the scalability and responsiveness of the system. In this paper, we present a scalable backend storage and retrieval architecture to support very large volumes of real-time measurements from wireless sensor networks. In particular, our contribution provides: (i) a database partitioning and structuring scheme coupled with a double-buffering technique to reduce the end-to-end delay while minimizing the processing power, and (ii) an optimized historical measurement data query format tailored for superior performance in terms of data retrieval responsiveness. Through a realistic emulator for large scale sensor network, we evaluate this storage and retrieval system to illustrates its delay and I/O benefits in both high and low traffic rate scenarios. The evaluation guides our design of an adaptive design, that applies batch insert method for smaller deployments to reduce insertion delay, and double-buffering for larger deployments to reduce I/O cost and avoid saturation, at the cost of higher delay.

INTRODUCTION

The cost of advanced sensor nodes (motes) is continually decreasing, rendering large-scale sensor networks deployment affordable and viable for many use cases such as precision agriculture. While these recent advances certainly raise many new opportunities and challenges, existing research has mainly focused on issues related to the scalability of data collection and routing protocols to reduce traffic overhead and thus increase the power efficiency.

Research on data-centric sensor management falls into

two main directions: streaming data; and historical storage. The streaming data approach considers the sensor nodes as stream producers and accesses data through query to the network. This approach targets real-time scenarios where only recent measurements are important; it combines real-time measurement aggregation and caching techniques to improve the performances in terms of responsiveness and accuracy [14, 18, 19, 13].

The historical storage approach takes the view that all measurements are valuable for later access and data mining, and consequently relies on centralized back-end storage and retrieval systems. The main challenge here is to conciliate high responsiveness for measurement queries with the overall I/O cost of system. There have been substantial research work on how to treat complex and expressive queries [5, 10] as sensor networks are scaled up (e.g., GSN [5]), with the introduction of historical storage of measurement streams.

Combining the streaming and historical approaches can provide a back-end system that supports both real-time streaming and historical data retrieval effectively. However, in a large-scale context, designing such system is compromised and no attempts has been done to our knowledge. To address this issue, this paper proposes a database design structure that conciliates the responsiveness for measurement queries with limited processing resources and the I/O cost resulting from different network configurations (in terms of number of sensor nodes and their reporting period) that may put different burdens on the back-end system. We propose two methods for supporting the wide range of measurement data volumes arising from different sized deployments: (1) batch insertion, and (2) double-buffering. The batch insert method consists of grouping the insertion of multiple data records by delaying the commit; it supports low input rate (low responsiveness) while ensuring low I/O cost or low insertion delay. The double-buffering method builds on the work in [4] by providing two separate buffers for receiving data from the network and for committing data to the database simultaneously, which significantly reduces delay over the batch-insert method. This improves the system processing capacities and thus greatly increases the data volumes inputted in the database.

SUMAC ARCHITECTURE

Our work in this paper is part of the SUMAC project [11]. The SUMAC project aims at developing an energy-efficient large scale wireless sensor network that can be effectively accessed and controlled remotely. A prototype has been developed and deployed in the campus of University College Dublin. The objective of this project is twofold: (i) research and design mechanisms and protocols to develop in-network processing capabilities able to aggregate measurements, react to alerts, change the nodes behavior. This effort is meant to develop a network middleware that supports application-aware network adapta-

Figure 1 : The SUMAC Architecture
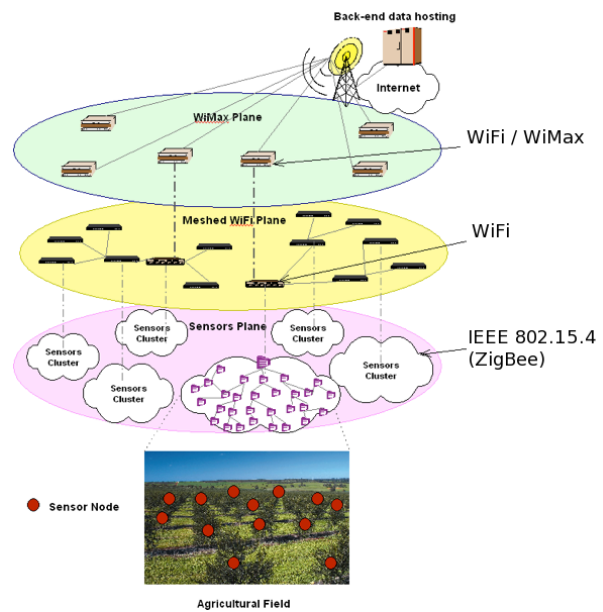
tions and network-aware application adaptations [12]; and (ii) a back-end system to enable web-based access and analysis of historical measurements. This back-end system includes all the design protocols and security mechanisms to convey the sensor measurement through 3rd party network; most importantly, it embeds intelligence to translate high-level policies into sensor network management procedures. In this paper, we focus on designing and developing a cost-effective storage and retrieval system able to meet the stringent demand of a large-scale wireless sensor network.

As illustrated in the Fig. 1, we use a wireless mesh network [17] to bridge together the geographically distant clusters of sensor networks and cover very large areas without incurring additional cost associated with wiring all cluster heads to Internet. The WiFi-based mesh network routes all data to few mesh portals that in turn aggregate and forward the sensor measurements to the back-end portal.

The SUMAC architecture mainly targets environmental monitoring. A typical example is vineyards [2], where the network can gather luminance, moisture, temperature or any other metric from several fields, and store them for historical purposes, such as future analysis of the conditions that favors an increase in the productivity. Considering this architecture, the amount of motes inside one single deployment can be significantly high. If we put one motes every ten meters to have a acceptable precision, we will have 100 sensors per hectare. This amount will reach quickly 2000 if we cover a typical vineyard of 20ha. The large scale support is then essential in such system.

Nowadays, the tendency is to dematerialize everything ; to store and process data on cloud such as Amazon EC2[1].

In such cloud computing systems, I/O operations, cpu cycles and memory usage cost. Thus, historical storage has to be optimized for I/O costs, and also for its scalability that can be represented as the size of the data input rate. There are two main approaches used to increase the scalability of a back-end system: (i) vertical scalability model that employs caching techniques, content partitioning, and other hardware-related techniques to improve the system serving capabilities, and (ii) horizontal scalability model that employs a highly-available cluster of computers where the load entailed by serving VOD services can be shared among different servers. While a computing cloud is capable of offering important horizontal scalability improvements, vertical scalability is an important design decision for our historical storage approach in order to reduce the operational cost.

RELATED WORK

Only few research works focused on efficient storage of large historical data volumes in sensor networks. However, historical storage performance has been relatively well studied by the data warehouse field. The basic concept employed to deal with high load flows is to trade off the data insertion delay, and thus deteriorate the responsiveness of the system to later data queries. The main idea behind this design is to streamline I/O access by transforming random I/Os into well structured sequential I/Os. The performance of such systems increases with the maximum delays constraints, which would further limit the responsiveness of the system putting an excessive lag between the effective data collection at sensor levels and their availability for real-time analysis. Some research works[8, 15] have already focused on this kind of management by combining efficient data structuring and buffering-driven smooth insertion algorithms. Although our research work falls within this broad research approach, we consider more strongly the responsiveness constraint that limits the extent of optimization at insertion level. As a consequence, our system calls for a dynamic insertion strategy that adapts to the offered load to keep an acceptable responsiveness performance.

Another mainstream research approach consists of using load-shedding[16], involving the reduction and aggregating of the incoming data stream to reach manageable load (measurements are averaged before storage). The research project HiFi[9] uses precisely this approach to accommodate measurement storage in sensor networks. This system is based on the TelegraphCQ[6] adaptive stream engine that includes a historical storage called OSCAR[7] which was designed to reduce I/O costs. Several types of algorithms are proposed in order to adapt the storage process to the offered load by using multi-resolutions load-shedding manipulations on disk. This approach is probably the best to adapt to the charge but it has the significant drawback of trading the measurement resolution for performances. We take the view that each single measurement generated by a sensor node is valuable for later analysis and data mining. We believe that much of the cost should be associated with the actual data collection at the sensor network level, and any received measurement data should be stored without reducing the fidelity/resolution of the measurement.

DATABASE DESIGN

The main objective of the SUMAC research is to achieve 'Scalability'. Yet, it is readily realized that one database cannot handle measurement data volumes stemming from several deployments of over 10,000 motes; considering a fairly low reporting period of 1 minute, the stored data volumes will be qualified in Terabytes within a year. Our system design needs to carefully take consider scalability factors to meet the challenge by using a hierarchical design element. In the following sub-sections, we will first present how we distribute the back-end storage and retrieval system to support several WSNs deployments; afterwards, we will see the mote-driven data structure partitioning in the DB in order to achieve higher query responsiveness.

Fragmenting across deployments

As using a single database does not scale well, we adopted a hierarchical design principle. Our overall storage and retrieval system is organized as follows:

- A single Master Database (M-DB) that contains information common to all deployments, network configurations, and user interface preferences. This is some sort of metadata DB;
- A Deployment Database (D-DB) for each deployment that manages a distinct sensor network, and stores all measurements related to this latter.

The M-DB is also used as a pointer to the different D-DB to help the receiving server to identify in which D-DB each measurement stream should stored. It is used in the same manner to retrieve data for an end-user connected through Internet. This structure has two main advantages: it is highly scalable because there is no limit on the number of sensor nodes supported by the system; also, we replicated the whole system, and assign several D-DB for a very large WSN deployment.

Partitioning across motes

In this sub-section, we introduce how the measurement data are stored in the D-DB in such way to maximize the performances of queries. The D-DB data has been designed to best accommodate the most common queries.

The most frequent queries that are used in monitoring applications have the following form: *Get the XX last readings of mote YY*. This means that the D-DB responsiveness will overly deteriorate with the number of motes being tracked. To address this we grouped the measurements by mote using the MoteID identifier to create different tables, in addition to a global time-stamping to historically nav-

Table 1 : Table schema for each measures_xx

Column:	timestamp	storedTime	count	raw values
Type:	TIMESTAMP	TIMESTAMP	SMALLINT	BIGINT
Indexed:	X			

igate within the collected measurements. The *raw values* column contains a binary version of all the different readings we get on each single mote active in the network. We use the BIGINT[1] in order to keep the disk usage at the lowest level possible.

A query for the last 20 temperature measurements of mote 42 will translate into the following *SQL* query:

SELECT sensorValue(values, 1) FROM measures_42 ORDER BY timestamp DESC LIMIT 20

Even with Terabytes of data we can always get the last readings in less than 100ms thanks to this index organization.

DATA INSERTION OPTIMIZATION

The database has been designed to have optimized performances for queries. Admittedly, this design implies that the places on disks are fragmented as well, requiring much more entry changes during sensor measurement storage, which limits the storage data rates and the responsiveness. In the following, we present and compare two methods we designed and adapted to have the best performances: batch insert method and double-buffer method. The batch insert method is based on the delayed commit that has been widely used by researchers and engineers due to its simplicity. The double-buffer method uses a double-buffer[4] to adapt the delay to increase the supported data rate further.

Batch insert method

The natural approach to deal with multiple insert in a database is to use *copy*-like[3] instructions; this, however, leads to insufficient performances in real-time, especially in the context of multiple spread tables. Unfortunately, in one sampling period (one second for example) we can have thousands of tables to modify. Another approach is to explicitly delay the commit of a constant period, so as to group together related updates and achieve efficiencies in terms of I/O. To efficiently perform this we designed two processes on the MeshServer: the first process receives sensor measurement packets and buffers them, while the second process communicates with the database to perform I/O operations. To create the delay, the database empties the buffer at overflow events or upon timeouts only, which ensures a bounded latency for sensor measurements storage.

This method provides a gain of supported deployment size from 3 to 4 compared to the conventional sequential storage solution. However, the main limitation resides on the

---

[1] BIGINT is a SQL type that is a 64 bits integer

Figure 2 : The double-buffer method

fact that it is not possible to ensure a high storage data rate. Due to the a-priori memory buffering and the usage of *insert*-like instructions, this method cannot support much more.

Double-buffer method

The main idea for double-buffering, which targets large networks, is to delete the two restrictions that arise with the above method. We first transform the memory buffer into an adaptable buffer inside a database that can be on disk. Then, we change this costly *insert* instruction into a *copy* one.

Figure 2 shows the stream line of the data storage manager with the two main buffers that are tables with the schema (moteid, timestamp, count, values) with absolutely no indexing. The MeshServer (MS) transfers after analysis messages to the first buffer while the DatabaseManager (DBM) performs a bulk insert inside final tables. When the buffer #2 is empty, the DBM asks exclusive rights to the buffer #1. When done, the roles are inverted, the DBM empty the buffer #1 while the MS fills the other and so on.

The bulk insert is done with *copy* instruction because we surely get more that one measure per sensor. Therefore the greater the ratio tuple/table is, the better the usage of this instruction is optimized. It also avoids buffer overflows, as it is written on disk. The next section details the performance analysis of the two methods we introduced here, and also discuss which method is more suitable for a given situation.
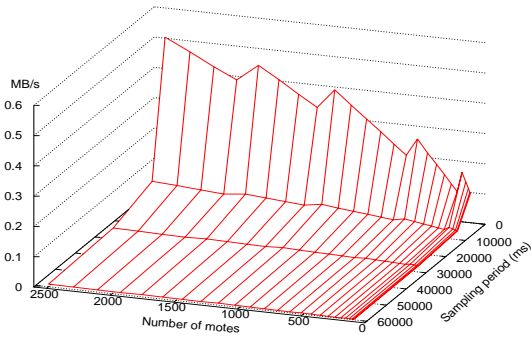
PERFORMANCE ANALYSIS

In this section, we will show the benchmark we deployed and its results. We have tested both methods in the same machine, on the same conditions in order to be the more precise possible. We will firstly present the emulator.

A word about the emulator

The emulator tries to reproduce reality by transforming the deployment, that we can see in the figure 1, into a data flow producer. It can provide the same behavior as the real network because it is built with the components from the mesh layer. Thus we can also simulate the usual MTU buffering inside the mesh network. Its timeout is configurable to fit with the end-user responsiveness requirements.

As to be expected, the offered load at the MeshServer increases proportionally to the number of motes and inversely to the sampling period. The MeshServer handles

Figure 3 : Offered load at the MeshServer

Figure 4 : Batch insert - MeshServer CPU usage

Figure 5 : Batch insert - MeshServer Memory usage

a traffic load of about 0.6 Mbps for a sampling period of 1 second and 2500 motes. While this received data rate is manageable, the entailed CPU and RAM usage is (see figure 4 and 5) since the MeshServer process each TCP packet to retrieve and interpret the encapsulated sensor packets. We can see the MTU buffering effects on the in-bandwidth on the MeshServer that we show on figure 3. Even though the increasing is linear in theory, we have a cut reality.

This emulator has many parameters as the size of deployments, counted as the total number of motes, the length of the sampling periods and the MTU buffering timeout. In order to have the accurate analysis, we deploy the three servers on one single machine[2] and the emulator on an other[3]. For the deployment we use a powerful tool called smartfrog that can deploy each component on whatever computer. In the real deployment we will also deploy our servers with this tool as it can interface with Amazon EC2[1] easily.
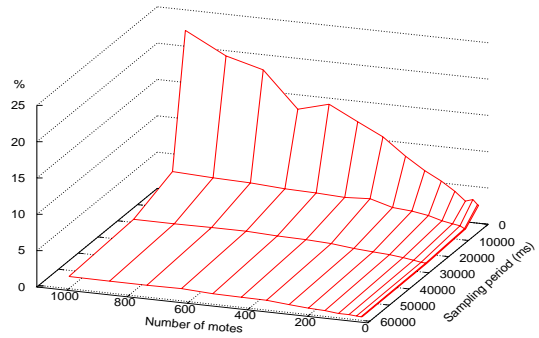
The validation of our solution uses the following metrics, which the emulator gathers via mostly Sun JMX:

- *CPU and Memory usage of the MeshServer and of the maintenance Daemon*: part of the usual cloud computing pricing policy. We need to control this metric to lower the final cost.
- *The Input/Output bandwidth rate*: represents data rate flow in the Internet. If this rate is too high we can consider splitting the single connection in two.
- *Input/Output cost on the storage disk*: is a critical indicator of performance, as as we do need to write down the readings on disk.
- *The delay upon insertion* which is equals to the difference between the timestamp upon insertion in the final tables of the database and the timestamp upon packet generation. It has an direct impact as the data are searchable through historical queries only if the data is inserted in the database.
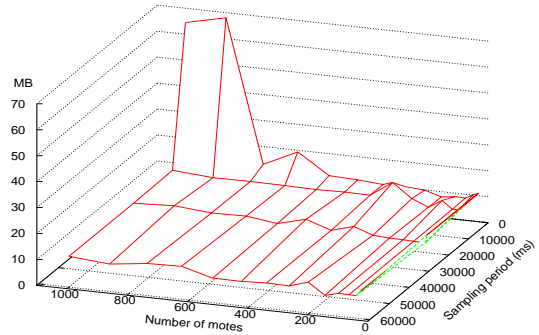
Having described the emulator parameters, the next section presents the simulation results.
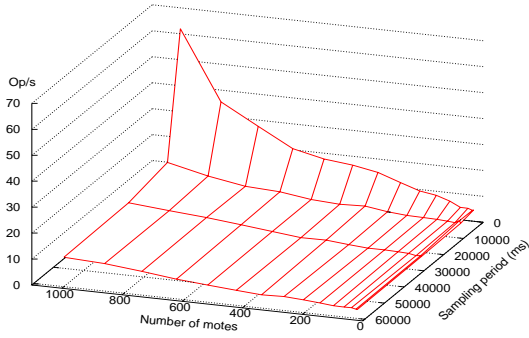
Batch insertion results

We first tried to test the limit of the system to see what to plot. We fixed the sampling period to 1s and then looked for the number of motes that made the system faulty[4].

To test the performance boundary of the historical storage system for a single deployment, we estimate that the limit on node membership is between 1000 and 1200 motes within a single deployment. Considering this limit, we have generated 60 test points, 15 on the number of motes axis and 4 on the sampling period one. As previous experiments have demonstrated, we have decided to have a quadratic repartition of those points in order to fit best the curves. Every metrics has been computed by doing the average among the time while being stable. The stable situation is declared after the table creation and a first insert has been done in the table. For this method, we will describe the plots with the ranges of [25-1000] motes and [1-60] seconds as sampling period.

A first significant result is the linear evolution of the CPU usage of the MeshServer, as shown in Figure 4. This computing time is spent mostly on dispatching each packet to the correct table. Even though we provide a cache solution based on hash tables to do a fast answer to this query, we still have a linear evolution. Unfortunately we have a high computational power needed: over 20% for 1000 motes.

The memory-usage represented in figure 5 is an exponential metric. Storing a useful cache as well as the temporary

---

[2]Pentium 4 3.2GHz, 1Go Ram, 2 HDs 5200rpm, PostgreSQL 8.3
[3]AMD64 2.6GHz, 1Go Ram

[4]We can measure if the configuration is not supported by looking at the shape of the plot timestamp $\rightarrow$ delay upon insertion
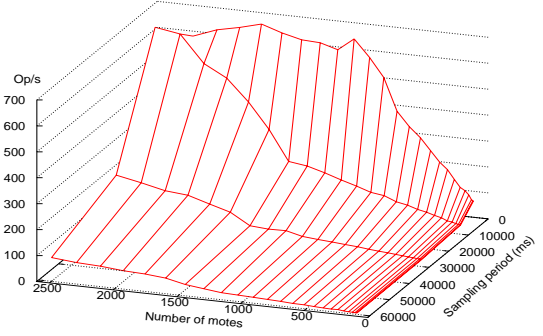
Figure 6 : Batch insert - I/O Write rate
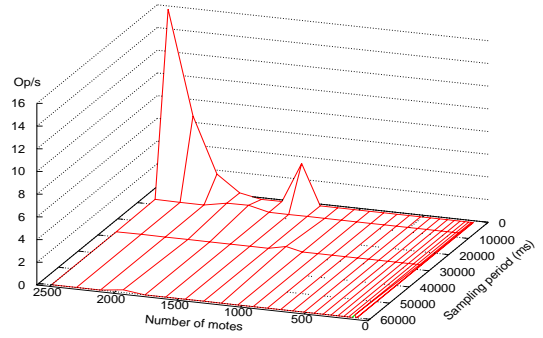
Figure 8 : Double-buffer - I/O Read rate
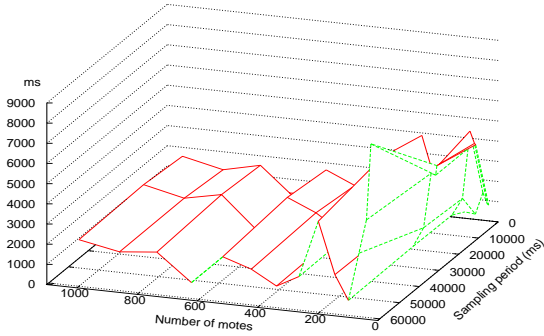
Figure 7 : Batch insert - Delay upon insertion

Figure 9 : Double-buffer - I/O Write rate

buffers increases memory costs. Even though the stack memory stays under 12MB, the heap can rise to 70MB. Because the cost of memory is comparatively cheap, optimizing memory usage is not our priority. In order to have a limited consumption, we implemented solutions of cross-deployment synchronization to empty the buffers effectively.

Between an elementary write command and the physical hard drive, there is always a cache solution in order to reduce frequent disk access. For instance, if we want to read 5 times in the same sector, we will read only once in the physical disk and hit the cache 4 times. A read operation in this method is not directly called. But the computation of the indexes needs reading in order to scan the index tree. The emulator shows us that almost no reading operations are performed as we almost hit every time this cache, ensuring low dependence on I/O read rate.

Considering figure 6, the I/O write rate suits an exponential curve that reaches the maximum ($\sim$700op/s) of the hardware on 1200 motes. Fortunately, the I/O cost is still pretty low for our requirements. We can here store a stream of 1000 motes that samples every second with only 60 operations per second. But as the evolution explodes exponentially it will be our limiting metric.

Figure 7 plots the measured delay upon insertion. As expected, the delay that remains between 0 and 11s, due to the MTU timeout (10s) and the batch timeout (1s). We can then correlate this figure with Figure 3, where both figures exhibit minima for the same configurations. An interest-

ing result is that the delay is independent of the sampling period.

In summary, the Batch insert method provides low cost for every metrics. However, it only supports smaller deployments of up to 1000 motes, which may quickly become a problem for deployments of scale. The second method in our design, the Double Buffer, aims at increasing the supported network scale at the cost of higher delay.

Double-buffer results
As before, we tested the limit of the system for a sampling period of 1s. We got a bound between 4000 and 4500 motes, which is already a huge evolution compared to before. We have indeed here a better performance with a ratio of $4 \sim 5$ over the deployment size.

Considering this limit, we vary the number of motes between 100 and 2500, considering four sampling periods as above. For performance issue and a better cache usage, we decided to put the buffer on one disk and the storage on an other.

We can see in figure 8 that the read rate stays low as the data flows in a *regular* rate. But when high rates are coming, the cache is not enough big and we have to swap to disk in a sequential order. Fortunately, this cost is insignificant compared to writing in an index input.

Figure 9 illustrates that even though the global range of delay values is higher than in the previous method, the double-buffer method is more scalable. Saturation occurs at the hard-disk capacity in a linear progression until 1000
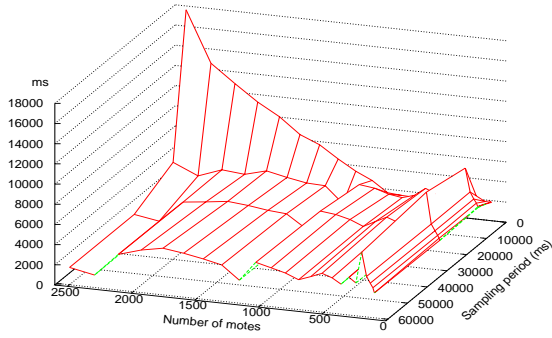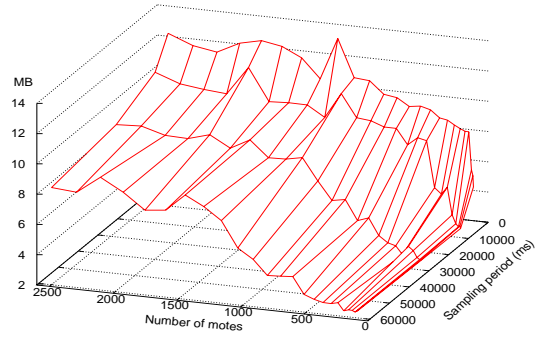
Figure 10 : Double-buffer - Delay
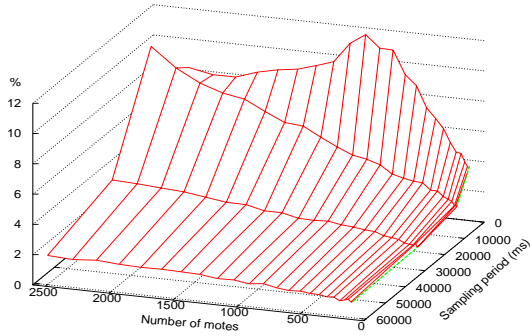

Figure 11 : Double-buffer - maintenance CPU usage

Figure 12 : Double-buffer - maintenance memory usage

	Batch insert	Double-buffer
Total CPU	$< 25\%$	$< 20\%$
evolution	$\mathcal{O}(t)$	$\sim \mathcal{O}(1)$
Total Memory	$< 70MB$	$< 50MB$
evolution	$\mathcal{O}(e^{Kt})$	$\mathcal{O}(t)$
Write rate	$< 100op/s$	$\max(700op/s)$
evolution	$\mathcal{O}(e^{Kt})$	$\mathcal{O}(1)$
Read rate	None	$> 10op/s$
evolution		$\mathcal{O}(e^{Kt})$
Delay	$< 11s$	$> 10s$
evolution	$\mathcal{O}(1)$	$\mathcal{O}(t)$
Max motes	1000-1200	4000-4500

Table 2 : Performance comparison

motes. Adding more nodes does not increase the cost any further. We can indeed support higher data rates with the same write rate. This double-buffer manipulation has the property to adapt itself to the charge it will take more time to do the flush with the same cost. If the load is too high, it takes more time to perform its operations. Therefore the current delay is increased and a the ratio tuple/table will rise on the next buffer switch, which will significantly help the bulk insert.

The double-buffer method thus trades off delay for scalability. Figure 10 shows the trend of this tradeoff. Here we can see that for high rates, we have a delay that can reach 18s. An important result is that the evolution seems to be linear. The delay we had in a deployment of 4000 motes is around 60s, so it scales well.

Figure 11 represents the evolution of the CPU usage on the maintenance process. We have seen in our measures that the MeshServer is not a bottleneck, as it only inserts into buffers so it is not displayed here. The maintenance (down in the first method) consumes a lot of power if the ratio tuple/table is low, we will use more resources to switch between tables. Therefore, we can see that the cost increases first, then stabilizes, and finally decreases.

Similarly, the figure 12 show the evolution of the memory consumption of the maintenance. Here we can see that we best control the memory pool. In this figure, we can see a logarithmic curve. The arguments on the CPU still remains, the better the ratio is, the less we use resources.

Method comparison

Both double-buffering and batch insert methods provide advantages and drawbacks, sometimes crucial. We need to separate the needs for every possible value of sampling period and number of motes. In order to identify the ideal method for each scenario, we need to split our domain of value between large and normal rates. We defines *normal* as: *supported by the batch insert method.* Higher rates have more stringent performance requirements which the batch insert method cannot handle.

In the batch insert method, we have correct overall performance for normal rates. We can ensure low I/O costs, fair CPU and memory usage, as well as a reasonable delay upon insertion. Therefore, we can choose this method for scenario that require low global cost. Unfortunately, due to its exponential explosion of the memory usage and the I/O cost, the batch insert method cannot support large rates.

On the other hand, the double-buffer method supports higher rates. Considering its capacity to auto-adapt to load, its behavior is controllable and does not saturate. Unfortunately, low rates are well supported but spawn a lot of I/O operations. The lowest recommended rate for this double-buffer method is the highest stable rate in the batch insert method.

Table 2 summarizes the different metrics and their evo-

lution[5] for a sampling period of 1 second. We can now easily see the ideal choice of method is highly scenario-specific. Moreover, this choice can easily change according to dynamic network membership.

Our software design can adapt methods in real-time. If a rate is lower than a certain bound (to be determined offline), the system uses the batch insert method. Otherwise, it switches to the double-buffer method. We demonstrate the use of this functionality through a practical scenario: a wine-grower owns 2 fields with 200 motes sampling at 1s. Our system is on the batch insert method by default. After some months, the wine-grower acquires 4 vineyards and instruments them. We have now progressively 800 motes that are joining the network. Our system can autonomously detect changes in the number of motes in each deployment and switch to the double-buffer method as the total number of nodes approaches 1000. The system executes this process automatically without any intervention or even awareness from the farmer, or more generally, the network operator.

CONCLUSION AND FUTURE WORK

In this paper, we introduced a new design for a storage and retrieval back-end system tailored to meet the requirements of a large-scale wireless sensor network. The objective is to build a cost-effective system with acceptable responsiveness characteristics to support real-time monitoring. The database has been structured with adapted data organization to meet the very specific pattern of measurement queries that are typical of sensor networks applications. Due to this particular design and the entailed performance requirements, we further investigated original optimizations to process large data volumes of measurements in real-time basis. We adapted the double-buffer technique to our system and show that the gain in performances is considerable. We evaluated the performance of our system through prototyping in our testbed. We argue that this technique may be fitted for any system that requires a large repartition over an important number of spread tables. It also provides further opportunities to design analytical models to compute the final cost considering on the cloud computing rates and the deployment size.

ACKNOWLEDGEMENTS

BIBLIOGRAPHIE

1. Amazon elastic cloud computing. *http://aws.amazon.com/ec2/*.

2. Camilie networks, wireless sensing for viticulture. *http://www.camalienetworks.com/*.

3. Postgresql 8.3 reference : Copy. *http://www.postgresql.org/docs/8.3/static/sql-copy.html*.

4. Wikipedia article - double buffering. *http://en.wikipedia.org/wiki/Double_buffering*.

5. Aberer, K., Hauswirth, M., and Salehi, A. Infrastructure for data processing in large-scale interconnected sensor networks. pages 198–205, 2007.

6. Chandrasekaran, S., Cooper, O., and Deshpande, A. Telegraphcq: Continuous dataflow processing for an uncertain world. *CIDR '03*, Jan 2003.

7. Chandrasekaran, S., and Franklin, M. J. Remembrance of streams past: overload-sensitive management of archived streams. *VLDB '04*, Jan 2004.

8. Chaudhuri, S., and Dayal, U. An overview of data warehousing and olap technology. *SIGMOD '97*, Jan 1997.

9. Franklin, M. J., Jeffery, S. R., Krishnamurthy, S., and Reiss, F. Design considerations for high fan-in systems: The hifi approach. *CIDR '05*, Jan 2005.

10. Gurgen, L., Labbé, C., Bottaro, A., and Olive, V. Sstreamware: a service oriented middleware for heterogeneous sensor data management. *ICPS '08*, Jan 2008.

11. Jurdak, R., Nafaa, A., and Barbirato, A. Large scale environmental monitoring through integration of sensor and mesh networks. *MDPI Sensors*, 2008.

12. Jurdak, R., Ruzzelli, A. G., Barbirato, A., and Boivineau, S. Octopus: Modular visualization and control for sensor networks. *Wiley Wireless Communications and Mobile Computing*, 2009.

13. Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. Tag: a tiny aggregation service for ad-hoc sensor networks. *OSDI 2002*, 2002.

14. Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. Tinydb: an acquisitional query processing system for sensor networks. *TODS '05*, 30(1):122–173, 2005.

15. Muth, P., O'Neil, P., Pick, A., and Weikum, G. Design, implementation, and performance of the lham log-structured history data access method. *VLDB '98*, Aug 1998.

16. Tatbul, N., Çetintemel, U., Zdonik, S., and Cherniack, M. Load shedding on data streams. *MPDS '03*, Jan 2003.

17. Union, E. Carrier grade mesh networks (carmen). *Misc*, Project Reference: 214994., 2007.

18. Yao, Y., and Gehrke, J. The cougar approach to in-network query processing in sensor networks. *SIGMOD '02*, 31(3):9–18, 2002.

19. Yao, Y., and Gehrke, J. Query processing in sensor networks. *CIDR '03*, Jan 2003.

---

[5]K is an undetermined constant