Optimizing network calls by minimizing variance in data availability times

Luis Neto[1], Henrique Lopes Cardoso[2], Carlos Soares[3], Gil Gonçalves[4]
{lcneto, hlc, csoares, gil}@fe.up.pt
Faculdade de Engenharia, Universidade do Porto, Porto, Portugal

[14]ISR-P, Instituto de Sistemas e Robótica - Porto, Portugal

[2]LIACC, Laboratório de Inteligência Artificial e Ciência de Computadores, Porto, Portugal

[3]INESC TEC, Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência, Porto, Portugal

***Abstract*—Smart Nodes are intelligent components of sensor networks that perform data acquisition and treatment, by the virtualization of sensor instances. Smart Factories are an application domain in which dozens of these cyber-physical components are used, flooding the network with messages. In this work, we present a methodology to reduce the number of calls a Smart Node makes to the network. We propose grouping individual communications within a Smart Node to reduce the number of calls is important to improve the efficiency of the process. The paper exposes and explains the Smart Node internal structure, formally describing the problem of minimizing the number of calls Smart Nodes make to Cloud Services, by means of a combinatorial *Constraint Optimization Problem*. Using two *Constraint Satisfaction Solvers*, we have addressed the problem using distinct approaches. Optimal and sub-optimal solutions for an actual problem instance have been found with both approaches. Furthermore, we present a comparison between both solvers in terms of computational efficiency and show the solution is feasible to apply in a real case scenario.**

Keywords–Sensor Simulation; Combinatorial Optimization; Time Synchronization; Smart Nodes; Industrial Wireless Sensor Networks.

I. INTRODUCTION

Wireless Sensor Networks (WSN) consist of sensors sparsely distributed over a given area to sense physical properties, such as luminosity, temperature, current, etc. They are composed of sensor nodes which pass data until a destination gateway is reached. Common applications are industrial and environment sensing, where they can be used to perceive the state of a machine and prevent natural disasters, respectively. Gateways in WSN play a preponderant role, since they acquire data from the sensors, do pre-processing and in more advanced cases are responsible to send the data to cloud systems for advanced processing. A Smart Node is a gateway that has enhanced data processing, reconfiguration and collaborative capabilities [1]. These components are nodes in Industrial Cyber Physical Systems, wich operate and control *Industrial Wireless Sensor Networks*. Considering a scenario that comprises a reasonable number of these components, in which:

- Gateways are in constant synchronization with Intra/Inter Enterprise Cloud systems.
- Gateways perform collaborative tasks by talking over the network.
- Human Machine Interface devices proceed to on demand requests to the Smart Nodes.

A large number of messages is expected generated by a large number of devices and services.

Gateways collect data from different sensor types (eg: humidity, current, pressure). These cyber-physical components are coupled to industrial machines, along with several sensors, which collect data about the operation of machines; finally, the data collected is treated and synchronized with Cloud systems for multiple purposes. The majority of sensors coupled to industrial machines sample data at very different rates and synchronize the collected data with the Smart Node, in the respective sampling frequency. A Smart Node can embed a set of different data treatment modules. These modules can be instantiated to provide different ways of treating sensor data in a way that can be represented as a graph (Fig. 1). A gateway internal logic arrangement is represented using a *directed acyclic graph (DAG)*. The graph structure in Fig. 1 can be divided into three levels, each with a different label and color assignment: the Sensor Level includes sensor instances (bottom level), providing data to the gateway; the data treatment level (middle level), includes nodes representing instances of algorithms embedded at the gateway that can treat information in several ways (eg: aggregate data using mean or other functions, perform trend analysis); the Network Level (top level) includes nodes where the flow resulting from the lower level nodes can be redirected to subscribing hosts in the network. This internal structure can be dynamically rearranged: new sensors and data modules can be loaded into the Smart Node; the connections between nodes can be reformulated to synchronize and treat data in new ways.

A problem of efficiency emerges due to the different rates at which the data is gathered from the different sensors. When the data reaches the Network Level nodes, it is immediately sent to the subscriber, a network Cloud service. Slight time differences in the availability of data lead the Network Level nodes to perform new and individual calls. If those time

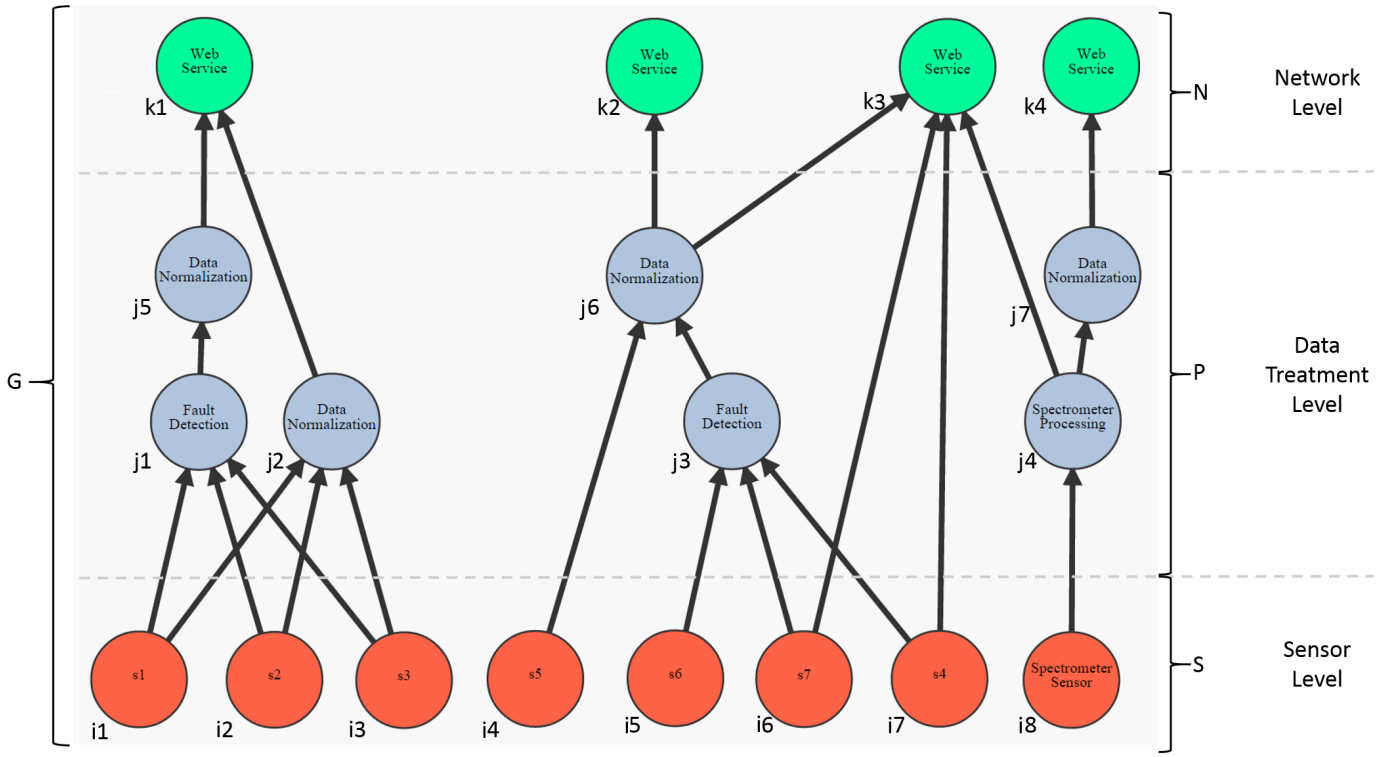

Figure 1. Internal Gateway configuration.

differences were eliminated, Network Level nodes would be synchronized and data from the different nodes could be packed together, reducing the total number of calls made and reducing the network traffic heavily. To accomplish synchronization among Network Level nodes, data buffers for all the edges connecting nodes previous to a particular Network node must be resized to compensate: (1) different time to process data by Data Treatment level nodes, since each module takes different time to process data; (2) different sampling rates of sensors, a same number of samples is accumulated at different times.

Taking advantage of the DAG representation of the gateway, we formulate and propose a solution to the problem as a combinatorial *Constraint Optimization Problem*.

In Section II, a formal definition of the problem is presented. Section III shows literature review, the problem formulation basis. In Section IV, the solving process is detailed along with assumptions, constraints and technology that has been used. In Sections IV and V, respectively, collected results and conclusions are presented.

II. Problem Definition

Each arc in the graph (see Fig. 1) has an associated buffer $b_{n,m}$. Given the fact that sensors are sampling at different frequencies $freq$, these buffers are filled at different rates. We define G as the set of nodes in a particular Gateway instance; three subsets of nodes are contained in G : $N \subset G$ is the subset of Network Nodes (index k nodes); $P \subset G$ is the subset of data Processing Nodes (index j nodes); $S \subset G$ is the subset of Sensor Nodes (index i nodes). The subsets obey to the following conditions:

$$G = N \cup P \cup S; N \cap P = \emptyset; P \cap S = \emptyset; N \cap S = \emptyset \quad (1)$$

Nodes in N can be classified as consumers; nodes in S are exclusively producers; nodes in P are both producers and consumers. Edges between nodes can be defined as:

$$e_{n,m} = \begin{cases} 1 & \text{if } n \text{ is consumer of } m : n \neq m; \\ & m \in P \cup S \text{ and } n \in N \cup P \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

As an example, we can observe in Fig. 1 that node j_6 consumes from i_4 (Sensor Level) and j_3 which is in same level (Processing Level) and all the k nodes (Network Level) only consume from inferior levels. To help in the definition of this problem, two additional subsets of nodes, containing the connections of a given node, are defined as follows:

$$W_n = \{j : j \in P \wedge e_{n,j} = 1\}, n \in N \cup P \quad (3)$$

Equation 3 defines a subset of nodes in P, which are producers for the given node $n \in N \cup P$. As an example (Fig. 1), for $n = j_6 : W_{j_6} = \{j_3\}$; for $n = k_3 : W_{k_3} = \{j_6, j_4\}$; and for $n = j_3 : W_{j_3} = \emptyset$ since it does not consume from any Data Processing nodes.

$$X_n = \{i : i \in S \wedge x_{n,i} = 1\}, n \in N \cup P \quad (4)$$

Equation 4 defines a subset of nodes in S, which are producers for the given node $n \in N \cup P$. In Fig. 1, these are the nodes i in the Sensor Level, from which Processing Level nodes and Network Level nodes consume. As an example (Fig. 1), for $n = k_3 : X_{k_3} = \{i_6, i_7\}$; for $n = k_1 : W_{k_1} = \emptyset$ since it does not consume from any Sensor Level node and for $n = j_6 : W_{j_6} = \{i_4\}$.
A processing node in P applies an algorithm to transform the data coming from its associated producers. The data generated at the sensor level is delivered to the processing nodes as a batch, which contains the number of samples equal to the size of the buffer for the corresponding edge.

In order to the processing to be possible, the number of elements in each collection must be the same. This constraint must be applied to the subsets W_n and X_n of a given node n in $N \cup P$, respectively; for that constraint to be respected, the size of every buffer associated to each element in $W_n \cup X_n$ must be the same. Formally this constraint can be represented as:

$$\forall n \in N \cup P, \forall m \in W_n \cup X_n : |b_{n,m}| = f(n) \quad (5)$$

Where $|b_{n,m}|$ represents the size of the given buffer for the given edge $e_{n,m}$ and $f(n)$ is the size of any buffer from which node n consumes.
The size of the buffer is adjustable and can vary from 1 to 1000. The objective of this problem is to arrange a combination of values to parameterize the size of every buffer $|b|$, for every arc in the graph, that minimizes the differences between times at the Network Nodes in which data is available to be sent to the network. To calculate the time that it takes data to be available at every node $k \in N$, the times for all its providers in the graph must be calculated. As data comes in collections (sets of single values), let us define *burst* as the exact time at which data is sent from one provider node to a consumer node and represent the *burst* of a node n as B_n.
The *burst* of a Sensor Node i, is defined by the product of its sampling frequency and the size of the buffer associated to the edge $e_{n,i}$ we are assuming. That way, every time a sample from a sensor is collected, that sample is sent to all consumers of that sensor. A *burst* of a Sensor Node to an adjacent consumer node m occurs when the buffer for the edge $e_{i,m}$ is completely filled, and is formally represented by the expression:

$$B_{i,m} = freq(i) \times |b_{i,m}| \times e_{i,m} = 1; \forall i \in S \wedge m \in P \cup N \quad (6)$$

For a Data Processing Node, the burst time must contemplate all the burst times from its providers, the time that it takes the associated function $T(f(n))$ to treat one data sample and the size of the buffer associated to the edge $e_{n,m}$ we are assuming. The expression which determines burst time for a Data Processing Node j to a consumer node m is defined as:

$$B_{j,m} = (\max_{i \in W_j \cup X_j}(B_{i,j}) + T(f_j) \times (|W_j| + |X_j|)) \times |b_{j,m}|; e_{j,m} = 1 \quad (7)$$

We assume that the growth in time complexity of the function $T(f_n) : n \in P$ is linear with the number of samples to process. Since the size of each producer buffer is equal, we multiply the total number of producers of j by the cost of treating a single sample. To calculate the *burst* for j_1 (see Fig. 1), we take the max *burst* of X_{j_1} and sum the product of $T(f_{j_1})$ (time to process one sensor sample) with the number of elements in X_{j_1} (which corresponds to the producers i_1, i_2 and i_3).
Finally, to calculate the *burst* of a Network Node $k \in N$:

$$B_k = \max_{i \in X_k \cup W_k}(B_{i,k}) \quad (8)$$

Using the expression to calculate the *burst* for each Network Node, the objective is to minimize the variance of *burst* for all the Network Nodes and minimize buffer sizes. By varying the size of the buffers in the graph, the variance of all burst times for Network Nodes and the summation of all buffer sizes are minimized. With a variance of zero or closer, data from different Network Nodes can be packed in the same payload and sent to the subscribers in the network. Even if the quantity of data exceeds the maximum payload size for the protocol or the physical link being used, the number of connections needed is far less than it is in independent calls. The number of buffers $|P \cup N|$, times an upper bound buffer size of 1000 is multiplyed by the variance. This way, the variance has more impact in the search of an optimal solution than the summation of all buffer sizes.

$$\hat{V}(B_k) \times \ 1000 \times |P \cup N| + \sum_{n \in |P \cup N|} \sum_{m \in W_n \cup X_n} |bn, m| \quad (9)$$

As follows from Equation 9, minimizing variance of burst times for network nodes is the major concern. To reflect this, variance is multiplied by the maximum possible size for a buffer (1000, which is a reasonable number of samples for a sensor), times the number of Processing and Network nodes. This will drive the solver to focus on a solution with less variance, and break ties by considering the minimal buffer sizes (as these incur a cost). With a variance of 0 for the bursts at the Network Level nodes, all data produced can be sent to the cloud using the same call. If variance is higher than 0, a threshold must be used to decide the maximum reasonable time to wait between bursts. In comparison with individual calls strategy – a call made every time a burst at the Network Level occurs – the number of calls to the cloud is minimized as a consequence. The theoretical search space of the problem is E^n, where E represents the total number of edges in the graph and $n = 1000$ is the ***Buffer Size*** domain upper bound. The real search space, imposed by the constraint of the Equation 5, can be determined by F^n, where $F = |P| + |N|$ is the total number of Processing and Network nodes in the graph.

III. Related Work

The theoretical background behind this problem has a large spectrum of application. The problem of modeling buffer sizes is mostly applied to network routing, where the works [2],[3] and [4] are examples. As we are not interested in dealing with networks intrinsic characteristics, those buffer optimization problems can hardly be extrapolated to this work. The domain of Wireless Sensor Networks (WSN) is another scope of application of buffer modeling optimization, with relevant literature in this domain; the section of *Routing* problems in [5]

covers a great number of important works regarding Flow Based optimization models, for data aggregation and routing problems. WSN optimization models care with constraints that this problem modulation does not cover, such as: residual energy of nodes, link properties, network lifetime, network organization and routing strategies.

A relevant work in WSN revealed to be of the major interest for this work. The authors presented and solved the problem of removing inconsistent time offsets, in time synchronization protocols for WSN [6]. The problem presented has an high degree of similarity with the case we are dealing. The problem is represented by a *Time Difference Graph (TDG)*, each node is a sensor, every sensor has local time and every arc has an associated cost time given by a function. The solution to the problem is given by a Constraint Satisfaction Problem (CSP approach. For every arc in the graph there exists an *adjustment variable* (analogous to the buffer size in this case), assignments are made to the variables to find the largest consistent subgraph, ie. a sub-graph in which inconsistent time offsets are eliminated.

Focusing the search in the literature domain of CSP problems, several works were revealed in the sub-domain of balancing, planning and scheduling activities that can be related to this application [7][8][9][10][11]. Namely, models of combinatorial optimization for minimizing the maximum/total lateness/tardiness of directed graphs of tasks with precedence and time constraints [7][11]. These problems are analogous to this work, and due to a simplified formulation with the same constraints (precedences and time between nodes), can be easily extrapolated to our case.

IV. Implementing and Testing

A. Problem Assumptions

The Smart Node application has several interfaces for real sensors, ranging from radio frequency to cable protocols. By testing this model with simulated scenarios, we assume no interference or noise of any type can cause disturbance in the sampling frequency. In a real case scenario, a sensor could enter in an idle state for a variety of reasons. In that case, data would not be transmitted at all, causing the transmission of data to the Cloud to be postponed for undefined time, waiting for the Network Level node burst depending on the idle sensor. For simplification, we assume a sensor never enters an idle state. Also, it is assumed that the time that takes to treat one collection of data will increase linearly for more than one collection, as mentioned for $T(f_j)$ when introducing Equation 7.

B. Constraint Satisfaction Problem Solvers

For comparison of performance purposes we implemented the problem using both *OptaPlanner* and *SICStus Prolog*. As the Smart Node is implemented in Java we can take advantage of a direct integration with *OptaPlanner* in future. On the other hand, we expected that *SICStus Prolog* would produce the same results with better computation times because of the lightweight implementation and optimized constraint library. Using this premises and the results presented in the next section a grounded decision about what solver to use in future implementations of the Smart Node can be made.

C. Tests

To validate the problem solutions several graph configurations were tested using the two implemented versions, based on *OptaPlanner* and *SICStus Prolog*, as described in Section IV. To test the implementations an algorithm to generate instances of the problem was built. The script generates instances of the Smart Node internal structure, DAG's, with a given number of Processing and Network nodes. Algorithm 1 briefly illustrates the approach:

Data: $G \leftarrow S \cup P \cup N$
Result: Smart Node internal configuration G
$notVisitedNodes \leftarrow G$;
$Pnodes \leftarrow randomInteger(\frac{|P \cup N|}{2}, |P \cup N| - 2)$;
$Nnodes \leftarrow nNodes - Pnodes$;
$Snodes \leftarrow$
$randomInteger(\frac{nNodes}{2}, nNodes + \frac{nNodes}{2})$;
$G \leftarrow S, P, N \leftarrow$
$generateNodes(Snodes, Pnodes, Nnodes)$;
$remainingEdges \leftarrow Pnodes \times 2 + Nnodes + Snodes$;
while $remainingEdges > 0$ **do**
  if $node \leftarrow notVisitedNodes.nextNode()$ **then**
    $notVisitedNodes.remove(node)$;
  else
    $node \leftarrow G.randomNode()$;
  end
  if *node is S* **then**
    connect to a random P or S node, disconnected nodes first;
    $remainingEdges - -$;
  else if *node is P* **then**
    get connection from a random P or S node, disconnected nodes first;
    connect to a random P or S node, disconnected nodes first;
    $remainingEdges - -$;
    $remainingEdges - -$;
  else
    get connection from random a P or S node, disconnected nodes first;
    $remainingEdges - -$;
  end
end

Algorithm 1: Smart Node instance generation.

Real scenarios generally have a higher number of Sensor Nodes, followed by a small number of Processing Nodes and an even smaller number of Network Nodes. Typically the total number of nodes does not exceed the 30 per operation. The generator picks aleatory numbers for the nodes bounded by a real case scenario application. Sampling frequencies for the sensors are assumed to vary from 400 to 2000 milliseconds. Functions to treat data in Processing Nodes are not typically complex. We measured the real case scenario functions to treat the minimum amount of data (1 sample) and we got values ranging from 0.19 to 0.38 milliseconds. To cover the buffer size domain, we need to take the worst case, 1000 samples. Given best and worst cases, the values attributed to cost of Processing Nodes ($T(f_j)$ in Equation 7) are between 1 and 40 milliseconds.

1) OptaPlanner: This solver [12] is a pure *Java* constraint satisfaction *API* and solver that is maintained by the *RedHat* community, and it can be embedded with the Smart Node application to execute and provide on demand solutions to our optimization problem. Because of the reconfigurable property of the Smart Node internal structure, each time the structure is rearranged, the solution obtained to the problem instance prior to the reconfiguration becomes infeasible. The integration (see Fig. 3) between the two technologies is accomplished by defining the problem in the OptaPlanner notation: (1) *Buffer Size* class corresponds to the *Planning Variable*, during the solving process it will be assigned by the different solver configurations; (2) *Edge* class is the *Planning Entity*, the object of the problem that holds the *Planning Variable*; (3) *SmartNodeGraph* class is the *Planning Solution*, the object that holds the problem instance along with a class that allows to calculate the score of problem instance. The score is given by implementing Equation 9; the best hard score is 0, which corresponds to null variance between the Network Levels nodes. The soft score correspond to the minimization of the summation of all buffer sizes and does not weight as much as hard score in search phase.

Since the search space is exponential, heuristics can be implemented to help the *OptaPlanner* solver to determine the easiest buffers to change. The implemented heuristic sorts the buffers from the easiest to the hardest. The sorting values are given by the number of ancestors of a given edge, an edge with a greater number of ancestors is more difficult to plan. Also, if an edge leads to a Network Node, it is considered more difficult to plan. *OptaPlanner* offers a great variety of algorithms to avoid the huge search space of most *CSPs*. These algorithms can be consulted in the documentation [13] and configured to achieve best search performances. For a correct comparison we used the *Branch and Bound* algorithm, which is the same algorithm that *SICStus Prolog* uses by default, without heuristics.

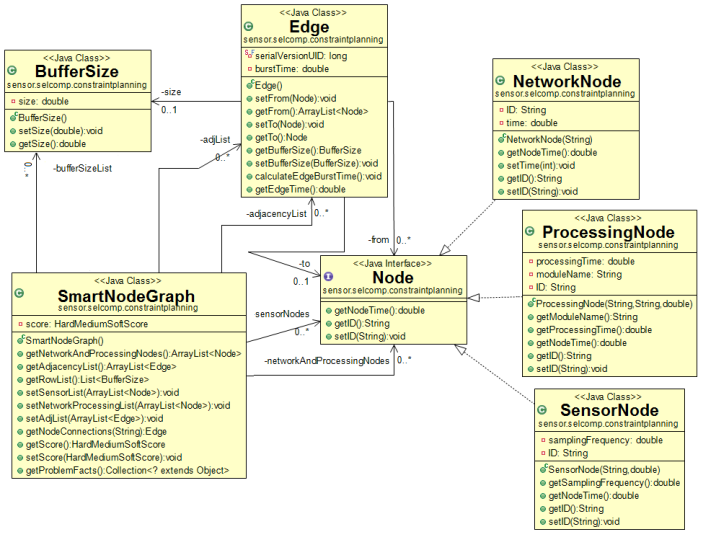

Figure 2. UML for Smart Node and OptaPlanner integration.

The UML diagram in Fig. 3 shows the modeling of the problem using the *OptaPlanner* methodology.

2) SICStus Prolog: *SICStus Prolog* [14] provides several libraries of constraints that allow to model constraint satisfaction problems much more naturally than the *OptaPlanner* approach, which follows from the fact that modeling a problem in *SICStus Prolog* takes advantage of the declarative nature of logic programming. The problem modeling involved four types of facts (to represent N, P and S nodes, and to represent edges) and six predicates (to gather variables, express domain and constraints). The *clpfd* (Constraint Logic Programming over Finite Domains) [15] library was used to model and solve the problem. This library contains several options of modeling that can be used to optimize the labeling process. In our case the labeling process takes as objective the minimization of the difference between the Network Node with the maximum burst time and the one with the lowest burst time (Equation 9). The variables of the problem are given by a list of all the facts *edge(from,to,buffer_size).*, where *buffer_size* are the variables to solve in a finite domain from 1 to 1000. In future implementations of the problem, global constraints and labeling options must be analyzed to ensure the modulation is the most optimized.

V. Results

For both implementations the results are shown in Tables I and II. The results shown, are a mean of 5 different problem instances, for each problem size (which is determined by $|P \cup N|$, see Section II. To gather results, the generator was used to generate 5 instances of the problem for each row. Then, both solvers were used in the same machine (Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz (8 CPUs), 2.5GHz, 16384MB RAM), with the same conditions (Windows 10 Home 64-bit), to run the tests. We established a limit of 60s, which was considered acceptable for the solvers to find a feasible solution in a real case. Another limit was the number of nodes used in the experiences. With a number of nodes in the order of 100, and a time window of 8h, both solvers were unable to give a response to most cases. Given the complexity stated, and given the fact that in real cases the number of nodes normally does not exceed 30, 50 nodes was the limit used for the tests.

OptaPlanner			
$\|P \cup S\|$	$\Delta V(B_k)$(ms)	$\sum \|b_{n,m}\|$	*Solution time(s)*
5	15.600	264.400	48.133
10	82.200	30.600	60.000
15	205.800	241.400	48.037
20	637.600	189.800	60.000
25	1494.000	56.600	60.000
30	1218.600	128.800	60.000
35	979.000	74.400	60.000
40	1434.400	74.600	60.000
45	1138.200	89.000	60.000
50	646.600	118.600	60.000

TABLE I. OptaPlanner results

The quality of the solutions found is mostly given by the second column, which represents the constraint of minimizing

SICStus			
$\|P \cup S\|$	$\Delta V(B_k)$(ms)	$\sum \|b_{n,m}\|$	*Solution time(s)*
5	0.400	731.800	38.022
10	1.800	738.000	48.894
15	73.000	1738.600	60.000
20	102.400	4912.600	60.000
25	600.000	8210.800	60.000
30	457.800	6214.800	60.000
35	351.800	7986.200	60.000
40	564.000	5792.200	60.000
45	630.000	10457.000	60.000
50	321.200	14706.400	60.000

TABLE II. SICStus results

the burst times at the Network Level. As we can see (Table II), the *SICStus Prolog* implementation shows the best results for the most relevant quality factor. In the third column, the summation of all the buffer sizes is lower in the *OptaPlanner* implementation (Table I). During the tests, we observed in the logs that the *OptaPlanner* was much more slower walking the search tree. Regarding all the columns, a clear tendency to worst results is obvious along the table, but in the last line of both tables, a sudden improvement in the variance occurs. This behavior enforces the NP-Completeness nature of this kind of problems. In every row of both tables in which a *Solution time* of 60 seconds is found, that row matches a sub-optimal solution. Since both solvers were programmed to stop at 60 seconds, mostly the solutions are not optimal. The sub-optimal solutions are feasible in a real case, even if the variance between call times is not zero, because the gap is heavilly reduced. The Smart Component can define a time window with the size of the variance, and this way, include all results in the same call.

VI. Conclusion and Future Work

Despite the search space of the problem, both solvers reached an optimal solutions in cases that are feasible to real application. In the future tuning options of the solvers must be explored. Another additional constraint to this problem could be the introduction of a case in which a single or several sensors are producing data with an higher priority. The problem can be easily reformulated to embrace that kind of situation modifying the objective function Equation 9. *SICStus Prolog* shows a clear advantage in computation time. That difference can be the reflex of the number of lines of code needed to model the problem. *SICStus Prolog* required eight procedures (predicates), against 10 classes and 1 *XML* configuration file for the *OptaPlanner* implementation. The difference in modeling complexity possibly causes an additional overhead. Another important remark is that, given the experience of implementing the problem and playing with the solvers options, two contrasts can be highlighted: (1) *SICStus Prolog* is very intuitive at the problem modeling phase, on the other hand, *OptaPlanner* required more effort, both in implementing an perceiving the methodology; (2) tuning the solvers, for example the time out feature that allows to stop the solver in the desired time, is more intuitive in the *OptaPlanner* approach. Considering all pros and cons, *SICStus Prolog* most probably will be chosen to integrate the Smart Node in future work. This experiments were made offline, as future work, the Smart Component can embedd the optimization code and adopt a strategy to optimize the variance in idle CPU time until an optimal solution is found online.

References

[1] L. Neto, J. Reis, D. Guimaraes, and G. Goncalves, "Sensor cloud: Smartcomponent framework for reconfigurable diagnostics in intelligent manufacturing environments," in Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on. IEEE, 2015, pp. 1706–1711.

[2] I. Ioachim, J. Desrosiers, F. Soumis, and N. Bélanger, "Fleet assignment and routing with schedule synchronization constraints," European Journal of Operational Research, vol. 119, no. 1, 1999, pp. 75–90.

[3] K. Avrachenkov, U. Ayesta, E. Altman, P. Nain, and C. Barakat, "The effect of router buffer size on the tcp performance," in In Proceedings of the LONIIS Workshop on Telecommunication Networks and Teletraffic Theory. Citeseer, 2001.

[4] K. Avrachenkov, U. Ayesta, and A. Piunovskiy, "Optimal choice of the buffer size in the internet routers," in Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05. 44th IEEE Conference on. IEEE, 2005, pp. 1143–1148.

[5] A. Gogu, D. Nace, A. Dilo, and N. Meratnia, Review of optimization problems in wireless sensor networks. InTech, 2012.

[6] M. Jadliwala, Q. Duan, S. Upadhyaya, and J. Xu, "On the hardness of eliminating cheating behavior in time synchronization protocols for sensor networks," Technical Report 2008-08, State University of New York at Buffalo, Tech. Rep., 2008.

[7] J. Błazewicz, W. Kubiak, and S. Martello, "Algorithms for minimizing maximum lateness with unit length tasks and resource constraints," Discrete applied mathematics, vol. 42, no. 2, 1993, pp. 123–138.

[8] B. Gacias, C. Artigues, and P. Lopez, "Parallel machine scheduling with precedence constraints and setup times," Computers & Operations Research, vol. 37, no. 12, 2010, pp. 2141–2151.

[9] K. Rustogi et al., "Machine scheduling with changing processing times and rate-modifying activities," Ph.D. dissertation, University of Greenwich, 2013.

[10] A. Malapert, C. Guéret, and L.-M. Rousseau, "A constraint programming approach for a batch processing problem with non-identical job sizes," European Journal of Operational Research, vol. 221, no. 3, 2012, pp. 533–545.

[11] J. H. Patterson and J. J. Albracht, "Technical noteassembly-line balancing: Zero-one programming with fibonacci search," Operations Research, vol. 23, no. 1, 1975, pp. 166–172.

[12] O. Team, OptaPlanne - Constraint Satisfaction Solver, Red Hat. [Online]. Available: http://www.optaplanner.org/

[13] ——, OptaPlanner User Guide, Red Hat. [Online]. Available: http://docs.jboss.org/optaplanner/release/6.3.0.Final/optaplanner-docs/pdf/optaplanner-docs.pdf

[14] M. Carlsson, J. Widen, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, and T. Sjöland, SICStus Prolog user's manual. Swedish Institute of Computer Science Kista, Sweden, 1988, vol. 3, no. 1.

[15] M. Carlsson, G. Ottosson, and B. Carlson, "An open-ended finite domain constraint solver," in Programming Languages: Implementations, Logics, and Programs. Springer, 1997, pp. 191–206.