DEBS Grand Challenge: Predictive Load Management in Smart Grid Environments

Christopher Mutschler[1,2]
christopher.mutschler@fau.de

Christoffer Löffler[2]
loefflcr@iis.fhg.de

Nicolas Witt[2]
nicolas.witt@iis.fhg.de

Thorsten Edelhäußer[2]
thorsten.edelhaeusser@iis.fhg.de

Michael Philippsen[1]
michael.philippsen@fau.de

[1] University of Erlangen-Nuremberg
Department of Computer Science
Programming Systems Group
Erlangen, Germany

[2] Fraunhofer Institute for Integrated Circuits IIS
Locating and Communication Department
Sensor Fusion and Event Processing Group
Erlangen, Germany

ABSTRACT

The DEBS 2014 Grand Challenge targets the monitoring and prediction of energy loads of smart plugs installed in private households. This paper presents details of our middleware solution and efficient median calculation, shows how we address data quality issues, and provides insights into our enhanced prediction based on hidden Markov models.

The evaluation on the smart grid data set shows that we process up to 244k input events per second with an average detection latency of only 13.3ms, and that our system efficiently scales across nodes to increase throughput. Our prediction model significantly outperforms the median-based prediction as it deviates much less from the real load values, and as it consumes considerably less memory.

Categories and Subject Descriptors

C.2.4 [**Computer-Comm. Networks**]: Distrib. Syst.—*Distrib. Applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*

Keywords

Distributed Event Processing; Hidden Markov Model.

1. INTRODUCTION

Complex Event Processing (CEP) and Event-based Systems (EBS) are the methods of choice for a near-realtime reactive analysis of data streams in many applications such as surveillance, sports, RFID-systems, stock trading, etc. [1].

The purpose of the ACM DEBS Grand Challenges is to create a framework to challenge event-based systems on real world data sets and to score solutions by application-specific evaluation criteria. In 2014 the focus is on distributed and scalable load monitoring and prediction in smart grid environments [2]. Data is collected from smart plugs in private households. Our solution is based on the *EventCore* event processing system that we originally developed to analyze position data in soccer games [3].

The remainder of this paper is organized as follows. Sec. 2 describes and formalizes the problem before Sec. 3 presents our contributions:

- We outline the key concepts of the *EventCore* middleware architecture and its internal processing (Sec. 3.1).
- We shed light on our solution and provide details of our efficient median calculation on data streams (Sec. 3.2).
- We show how to address data quality issues (Sec. 3.3).
- We present our enhanced prediction model that is based on a hidden Markov model derivate (Sec. 3.4).

Sec. 4 evaluates our system implementation on the real world smart grid data set. Our solution performs well and meets the grand challenge evaluation criteria. Our solution efficiently processes thousands of events per second with low latency. It scales across several machines. Its prediction model forecasts loads much more accurately with less memory than the proposed median-based prediction. Sec. 5 reviews related work before Sec. 6 concludes.

2. PROBLEM DESCRIPTION

This year's challenge seeks to demonstrate the applicability of event-based systems to scalable, real-time analytics of high volume sensor data. The scenario originates from the smart grid domain and addresses the analysis of energy consumption measurements. This section describes the data set (Sec. 2.1) and the requested queries (Sec. 2.2).

2.1 Data Description

The data is synthesized from real-world profiles collected from a number of smart-home installations in Germany. In the hierarchically structured data houses denote the topmost entities and consist of several households, each of which has several smart plugs. Each smart plug provides two measurements: the current load in Watt (W) and the work accumulated from the start (or reset) of the sensor in kilowatt hours (kWh). A data element in the input stream is a 7-tuple:

(id, ts, value, property, plug_id, household_id, house_id).

where *id* identifies a measurement, *ts* is its time stamp, the float *value* is the measurement, the *property* defines the type of the measurement, i.e., either load or work, *plug_id* identifies a plug in *household_id* of *house_id*.

DEBS '14, May 26–29, 2014, Mumbai, India.

In total 2,125 plugs were installed in 40 houses. Data acquisition started Sept. 1, 2013 00:00:00, ended Sept. 30, 23:59:59, and yielded about 4,055 million data entries.

2.2 Query Description

The grand challenge asks for two queries: (1) a short-term load forecast, and (2) a load statistics evaluation.

For individual plugs and houses ***Query 1*** forecasts the future load based on both the current load measurements and a model which is trained on historical data. Such a forecast can be used to adapt the energy production levels. The model is suggested in the Grand Challenge specification.

The model breaks the full acquisition time into N slices $s_i, i \geq 0$, of $|s|$ seconds length each. Hence, s_i spans the time interval $s_i = [ts_{s_i}^{start}; ts_{s_i}^{end}]$ or $[t_{start}+i\cdot|s|; t_{start}+((i+1)\cdot|s|)-1]$, where t_{start} is the time stamp of the first measurement. Whenever a plug sends an event $e_i \in s_i$ the model predicts the average plug load $\hat{L}$ on the upcoming slice s_{i+2} as

$$\hat{L}(s_{i+2}) = \frac{1}{2} \cdot (L(s_i) + median\{L(s_j) | j = i+2-n\cdot k\}),$$

where $L(s_i)$ is the average load of slice s_i. If the data of $n \in \mathbb{N}^+$ previous days is in the data set, with k as the number of slices in any 24 hour period, the second term is the median of the loads of all previous slices with the exact same time of day. To predict loads of homes, the averages of all their plugs are sumed up.

The query applies time slices of 1, 5, 15, 60, and 120 minutes length. The format of the plug-based output is

(ts, house_id, household_id, plug_id, predicted_load),

where *ts* is the start time of the slice and *predicted_load* is the value $\hat{L}$. For houses output events are encoded as

(ts, house_id, predicted_load).

Query result events are to be generated every 30 seconds.

Query 2 finds outliers in the energy consumption. For each house the query yields the percentage of plugs whose median load is above the median load of all installed plugs.

Whenever a plug sends an event the query generates two outputs, one for a sliding window of one hour and one for a window of 24 hours. If the *percentage* of outliers remains unchanged, the output is suppressed. Output is formated as

(ts_start, ts_stop, house_id, percentage)

where *ts_start* and *ts_end* are the time stamps of the first and last plug events in the window.

3. SOLUTION

This section describes the key ideas of our solution, our data quality handling, and our enhanced prediction model.

3.1 EventCore Middleware Architecture

The *EventCore* event processing middleware provides a scalable publish/subscribe infrastructure for loosely coupled event-based software components and manages event dissemination between event detectors, i.e., instances of stream operators. Fig. 1 illustrates the basic architecture. Detectors receive events either from the network or from other detectors that run on the same machine. When the middleware receives an event it inserts it into a lock-free ring-buffer.

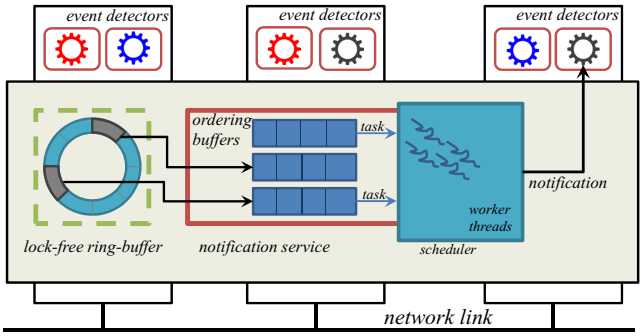


Figure 1: EventCore architecture.

Compare-and-swap techniques allow simultaneous read and write operations. There is one reordering buffer per detector that emits events to the detector as soon as a correctly ordered event stream is guaranteed. The middleware uses a scheduler and a thread pool to execute the functional part of the detector that processes the events.

For the queries of this grand challenge we implemented several event detectors that work on a per-plug and a per-house basis, see Sec. 3.2. Depending on the measurements in the streams, they are dynamically instantiated and configured on demand. The middleware does not have access to the semantics of the dynamically plugged-in detectors but only communicates via events.

3.2 Query Implementation

To achieve the required output we split the queries into several parts, see Fig. 2. We load the `PLUG_VALUE` data on a machine and disseminate the streams over the network.[1] There is one `Complete Median` detector running in the system. A `Fabric` event detector runs on every machine. It adaptively creates instances of the `Prediction` and `Outlier` detectors (depending on the actual plug and house ids that occur in the input). There are at most 40 instances of the `Outlier` and `House Prediction` detectors (one per house) and at most 2,125 instances of the `Plug Prediction` detector. Each detector is configured to handle its particular subset of plug values. The `Plug Prediction` and `House Prediction` detectors generate the Query 1 results, see Sec. 3.2.1. The `Outlier` detector produces the Query 2 results (using `MEDIAN` and `PLUG_VALUE` events), see Sec. 3.2.2.

3.2.1 Query 1: Load Prediction

For the load prediction query we combine two operators in the `Plug Prediction` detector: (1) a window-based filter that collects measurements from a plug/house to deduce the average load value for the current slice, and (2) a prediction operator that takes the average and the median of old averages to predict the load of a future slice.

The first operator is a simple $O(1)$ average filter that sums up the loads in the current slice and divides this value by the total number of measurements in that slice.

The second operator holds a history of the previously derived averages for a given plug and collects the set of slices that are relevant for the prediction whenever a new slice starts. The list of averages and also the median remain fixed for the duration of the slice.

[1] If we have fast hard disks we may also load the file directly into memory and then only disseminate the time stamps.

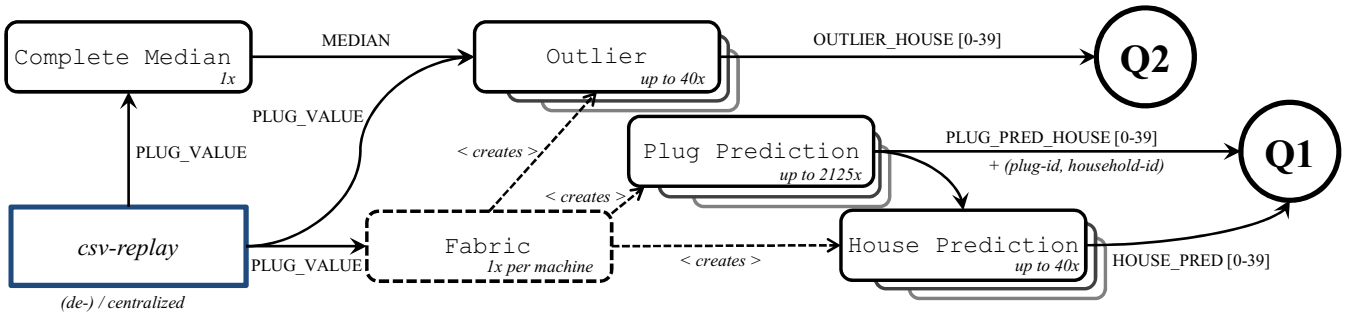


Figure 2: Query implementation and operators/event detectors.

To derive the median efficiently we implemented an adaptive histogram-based filter that works on a sliding window and that uses the two data structures shown in Fig. 3.

The upper data structure is a FIFO-based ring-buffer that stores the load values measured over the last $|s|$ seconds. In Fig. 3 at time ts the loads 18.53 and 16.26 were received. New load values are added on the right hand side of the buffer. As soon as the time stamps of the oldest measurements are below t-$|s|$ they are dropped on the other end.

The second data structure is a binary AVL-tree-based indexing structure [4] sorting the load values in the buffer. It is updated whenever values enter or leave the FIFO-buffer. Each element has both a load value and a histogram counter. The counter tells how often its load value appears in the current window. Whenever a load value is inserted or deleted, starting from the median element a binary search finds the correct position in the tree to update the counter. The median element always has a counter of 1, and possibly existing nodes with equal load exist on the left and the right side of the median. If the load value does not yet exist we insert a new node. If a counter is decremented to 0 the node is removed from the tree.

Because of the binary AVL-tree structure the algorithm has a complexity of $O(\log n)$ for look-up, insertion, and deletion. This includes rebalancing and the potential updating of the median pointer. The median itself is queried in $O(1)$.

The per-house prediction subscribes to all relevant plug prediction events. As the average load $\hat{L}(s_i)$ is calculated analogously to the averages of the plug, we can simply sum up the particular plug predictions and publish a single event for the load prediction of all the plugs in a house.

3.2.2 Query 2: Outliers

The outlier query consists of three operators: (1) a complete median that calculates medians for 1- and 24-hour windows over all plugs (implemented in the `Complete Median` detector), (2) a per-plug median that calculates running medians for each plug over 1- and 24-hour windows, and (3) an outlier detector per house that identifies the percentage of plugs with a median load higher than the median of all plugs for 1- and 24-hour windows. Operators (2) and (3) are implemented in the `Outlier` detector.

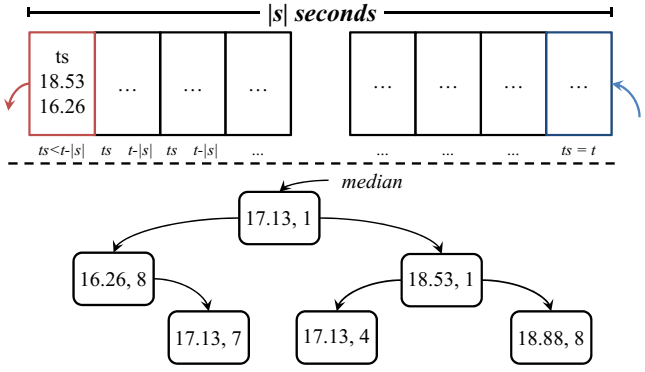


Figure 3: Adaptive median algorithm.

The adaptive histogram-based median filter from Query 1 cannot be used for (1) because of its $O(\log n)$ complexity. The number n of window updates may be too large if we replay the data faster than real-time, or if we increase the sensor data rate, the number of plugs, or the size of the windows. To speed things up we employ a second median filter with constant complexity on insertion, deletion, and median querying that trades memory for speed.

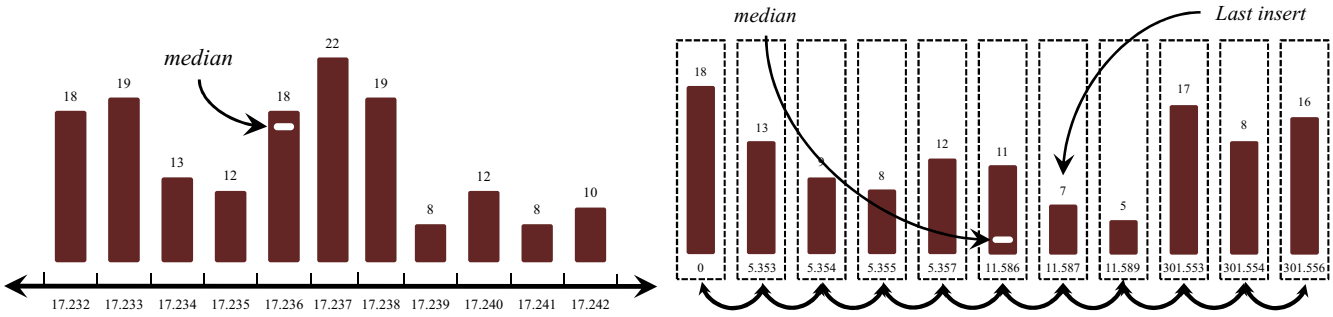
Under the assumption that a plug only has a limited range of possible loads (electric sockets are usually limited to 3,680W) we allocate an array of 3,680·1,000 indices to store the histogram of load measurements (with 3 decimal places). Fig. 4(a) illustrates the basic principle. The bars represent the histogram counter that are stored in the slots of the array. In Fig. 4(a) a load of 17.240W has occurred 12 times within the current sliding window. The median is encoded as a combination of the index value, i.e., the load, and the height in the histogram bar, i.e., the position among a number of equal load values. As in the AVL-tree we keep track of the number of values to the left/to the right of the median and update the position of the median accordingly.

For the 24 hour sliding window it takes 700 MBytes of data, plus the necessary time stamps that use less than a megabyte to store around 183.8 million measurements. But this buys the lowest possible and constant complexity on insertion, deletion, and median querying.

Operator (2) needs yet another median algorithm. The one from above consumes too much memory as is would need up to 2,125 arrays. The median algorithm of Query 1 can be made faster by exploiting the fact that subsequent plug measurements do not differ much. Hence, instead of the general purpose $O(\log n)$ search strategy on an AVL-tree we use a doubly-linked list as the seconds data structure, see Fig. 4(b). From the index of the last insertion of a value, a linear scan for the value/counter that is responsible for the next inserted value significantly outperforms logarithmic insertion.

Operator (2) takes about 600 KBytes per plug plus the dynamically allocated histogram.

As operator (3) only stores the information provided by the first two operators its memory consumption is negligible, and as operator (3) only derives percentages over the previously generated events its complexity is constant.



(a) Fast median algorithm. (b) Doubly-linked list median algorithm.

Figure 4: Median algorithms.

3.3 Handling Data Quality Issues

As the plug's data transmission is sometimes interrupted the streams may have gaps. This is obviously a problem as the prediction of $\hat{L}(s_{i+2})$ needs the $L(s_i)$ stream. But the predictions of $\hat{L}(s_{i+2+n\cdot k})$ also need the stream values of previous days. Missing data can cause miss-predictions. Between time stamps 15 and 26 in Fig 5 the transmission is interrupted (grey area). Both the load (dashed line) and the work (dotted line) values are missing/lost. As the plugs continue to measure the accumulated work even in a gap it is possible to reconstruct the missing loads when transmission is back, at least approximately.

To cover such gaps in the sensor data streams we employ additional event detectors that compensate the missing data. There is an operator for every plug's data stream that permanently reconstructs missing load values and that transparently inserts them into the data stream for subscribing detectors. Hence, the subscribers do not even recognize that there was a gap in a plug's data stream.

For the approximation we first reconstruct the work before (w_1) and after (w_2) the gap. We achieve this by adding the aggregated load values between the last work update and the beginning of the gap to the current work value. The actual work value at the end of the gap is calculated vice versa as soon as the work value is updated later on. The average load during the gap can be derived from the linear approximation of the work as l_{avg}=(w_2-w_1)/(t_2-t_1).

But for reconstructing the missing load we do not use a linear approximation but instead try to mimic a switching behavior. Under the assumption of (at most) a single switch this switch needs to be at time stamp

$$t_s = t_1 + \frac{(t_2-t_1)\cdot(l_{avg}+l_2)}{l_1-l_2}$$

to achieve the average load. The detector calculates t_s at the end of the gap, and sends the appropriate correction events

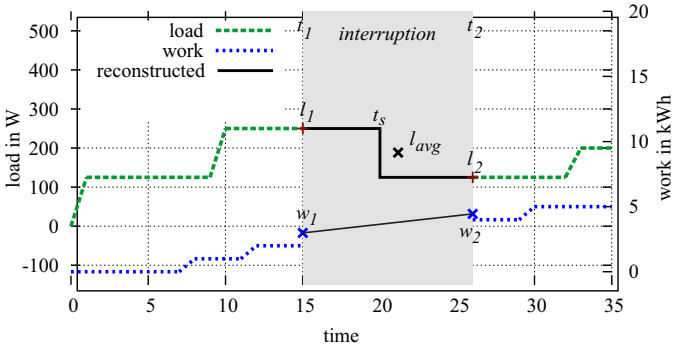


Figure 5: Interrupted data transmission.

so that other detectors can reconstruct their load measurements, i.e., their history appropriately. If the work value w_2 after the gap is below w_1 the sensor must have been reset. Then we do not attempt reconstruction.

3.4 Enhanced Prediction Model

The median-based load prediction model as suggested in the Grand Challenge description can be improved for more accurate forecasts. For Query 1 we propose to use a variant of a hidden Markov model (HMM) instead of the suggested model. Our HMM no longer breaks the measurement time into time slices but instead predicts the loads continuously.

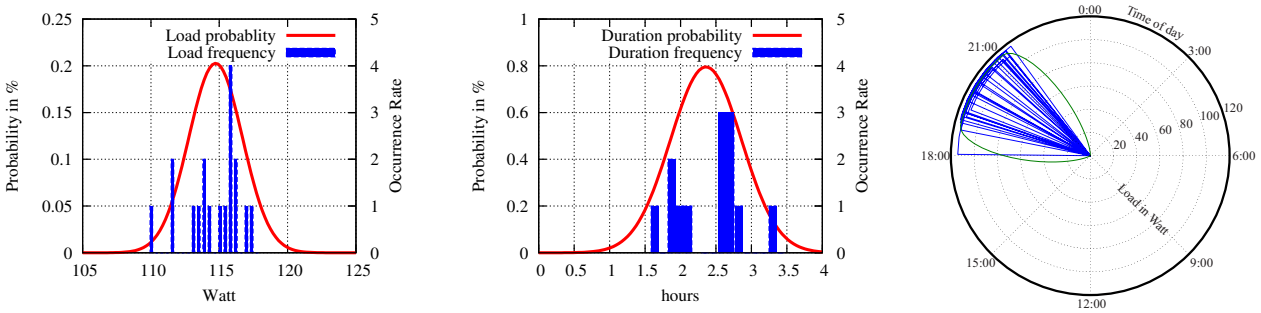
The hidden states of the HMM describe the behavior of the consumer that is connected to a particular smart plug. Assume there is one consumer per plug. To model energy consumption and hence the load of a plug there are three parameters: (1) the average load, (2) the duration of use, and (3) the *turn-on-time* on a day. We model (1) and (2) with a parameterized normal distribution, and (3) with a cyclic *von Mises-Fisher* distribution (defined by a distribution center and a concentration factor), see Fig. 6. A detector extracts the information from the data streams and uses it to train the HMM over one week of runtime. For better sequence modelling, an additional event detector converts the raw sensor data into a sequence of consumer events $S \in \{C_1, ..., C_j\}$ that are then used to train the HMM.

Because of its simple left/right-architecture [5] we can only step strictly forward through the HMM. But we may start and terminate in each of its states that continuously emit predictions for the average load, the average duration, and the turn-on-time of its energy consumers.

After its training the HMM detector predicts loads by first calculating the state probabilities $P(s_i|S,t)$, i.e., the probabilities for being in state s_i at time t for an observed sequence $S \in \{C_1, ..., C_j\}$. The prediction component then derives the most probable load for the prediction time t' by propagating the current weighted state probabilities through the model's states. This is achieve by multiplying them with the transition probabilities that are based on the distributions of the duration and turn-on-times. The state s_i that maximizes the probability predicts the load for time t'.

4. EVALUATION

For the evaluation we load the `csv`-file with the sensor data on one machine (in chunks of 100 million entries) and synchronize the processing by disseminating time stamps from a dedicated synchronization machine. The event processing machine uses a Core i5 750 CPU @ 2.67 GHz and 4 GB of main memory. Our network is a Gbit LAN.



(a) Distribution of load. (b) Distribution of duration. (c) Distribution of turn-on-time.

Figure 6: Average load, duration, and turn-on-time distributions per consumer (house 33, h'hold 1, plug 2).

Sec. 4.1 and 4.2 show results using the baseline prediction model before Sec. 4.3 evaluates our enhanced prediction.

4.1 Throughput Evaluation

We measure the throughput by increasing the speed of streaming the sensor data into our event processing system (without overloading it). Tab. 1 shows the throughput measurements per query and workload as requested in [2]. We start the event processing systems and deactivate the processing for the particular houses in order to achieve the requested workload measurements. The baseline, i.e., the lowest throughput, is used to derive the relative throughput gain for the other configurations. For instance, Query 1 runs 3.69× faster on 10 instead of 40 houses at a time.

For Query 1 we achieve the best throughput with a workload of 10 houses (we replay the data 300× faster than real time). The average is 244k events/sec. and the 10th and the 90th percentile only differ by 2k and 1k events/sec. If we increase the workload, i.e, the number of processed houses the adaptively generated additional event detectors consume more CPU power which slightly reduces the throughput.

As Query 2 consumes more CPU power (there are more medians and operators) the throughput is slightly lower. However, we still process 220k events/sec. with a workload of 10 houses and 62k events/sec. on the whole data set.

We also run our system with all event detectors activated. In that case the average throughput is 123k events/sec. with 10 houses and 33k events/sec. with the full workload, i.e, still 33× faster than single speed/real time.

When we use two event processing machines for the `Outlier` detectors of half the houses each and one event processing machine for the `Complete Median` we reach a total throughput of 135k events/sec. on average (133k and 140k for the 10th and 90th percentile) to process query 2 which is 2.18× better than on a single machine.

Table 1: Throughput (1000 events/sec.), 2 workers.

Query 1	10%	avg.	90%
10 houses	242.0/3.67×	244.0/3.69×	245.0/3.70×
20 houses	131.0/1.99×	131.0/1.98×	132.0/1.99×
40 houses	65.8/1.00×	66.1/1.00×	66.2/1.00×
Query 2	10%	avg.	90%
10 houses	219.0/3.54×	220.0/3.55×	220.0/3.54×
20 houses	123.0/1.99×	123.0/1.98×	124.0/1.99×
40 houses	61.8/1.00×	62.0/1.00×	62.1/1.00×
Queries 1+2	10%	avg.	90%
10 houses	122.9/3.72×	123.0/3.72×	123.5/3.72×
20 houses	65.8/1.99×	66.0/2.00×	66.3/1.99×
40 houses	33.0/1.00×	33.0/1.00×	33.2/1.00×

4.2 Latency Evaluation

In general we reach latencies well below 100ms, see Tab. 2. The latency of Query 1 is a little higher because of the large number of event detectors our system needs to manage. The overhead of event detectors is negligible for the performance of Query 2. The `Plug Prediction` detectors have very low complexity and hence very low latencies on average. The latencies for running Query 1 and 2 together are similar as the operators do not depend on each other.

4.3 Prediction Model Evaluation

To evaluate the prediction model we took a specific plug (house 33, household 1, plug 2) and used week 1 of the data to train our HMM. We then configured a prediction time of 120 minutes, i.e., the largest window that is used by the initially proposed `Plug Prediction` detector. Fig. 7 shows the load in Watts over 72 hours (3 days) in week 2.

The results show that both the median-based and the HMM-based prediction forecast the load very well. However, the HMM-based prediction is slightly better, as it forecasts the load correctly on the first try. In contrast, the median prediction often shows a few steps before it reaches the correct load value, see the second large peak for instance. This is because the turn-on-time varies on previous days. Similarly, notice the steps of the median prediction after the last peak. While the HMM forecasts 0 load correctly earlier.

Our HMM-based prediction only consumes a few float values to parameterize the distributions and hence scales for a large number of plugs. The runtime performance of the HMM prediction is similar to that of the median-based prediction (the HMM must calculate a limited number of state probabilities per time unit and the median prediction must derive the median from a limited number of historical averages). Thus although it is more efficient with respect to

Table 2: Latency measurements in milliseconds.

Query 1	10%	avg.	90%
10 houses	4.3/1.00×	13.3/1.00×	15.1/1.00×
20 houses	6.9/1.60×	22.5/1.69×	41.3/2.74×
40 houses	12.5/2.90×	76.2/5.73×	193.0/12.8×
Query 2	10%	avg.	90%
10 houses	2.9/1.00×	11.0/1.00×	9.8/1.00×
20 houses	6.0/2.07×	15.8/1.44×	24.4/2.49×
40 houses	11.2/3.86×	27.0/2.45×	45.4/4.63×

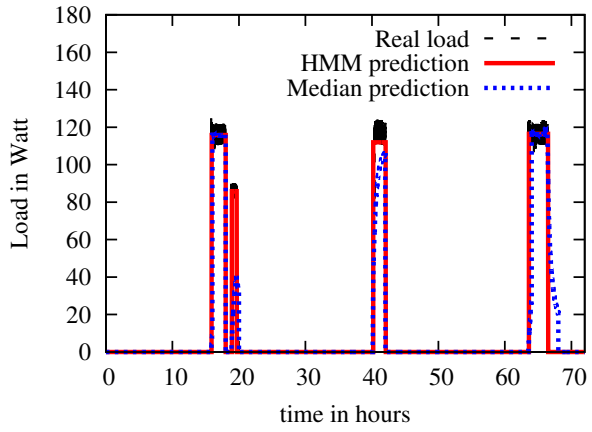


Figure 7: Load prediction.

resource consumption, the HMM prediction also slightly outperforms the median-based prediction model.

5. RELATED WORK

We first describe related work on online median filtering before we give an overview of smart grid load forecasting.

There is numerous work that deals with the efficient calculation of medians. However, most of the existing work cannot be used for data streams as it is either batch-oriented or because it only provides imprecise results [6, 7]. Distributed algorithms [8] may scale across threads and machines but are still no valid option for large windows. Instead, we adopted general purpose algorithms and derived the medians efficiently for each of their specific tasks.

Although the technical abilities and efficient communication infrastructures that allow data stream analysis in smart grids are relatively new [9], work on load forecasting and outlier detection dates back to the early 90s [10]. Artificial neural networks (ANN) [11] have been very popular back then before Hippert et al. [12] analyzed the ANN approaches and identified serious lacks of testing completeness. Other recent methods to forecast load or to identify outliers are Support Vector Machines [13] or linear discriminant analysis (LDA) [14]. For this paper, we adopted hidden Markov models that are successfully applied to other several smart grid challenges [15].

6. CONCLUSION

Our solution to the DEBS Grand Challenge is based on a scalable publish/subscribe-based middleware infrastructure where new operators are created on demand. Our approach scales across several machines to achieve higher throughput while keeping the latency small.

We also went extra miles and showed how to handle data quality issues by reconstructing missing measurements. We presented our prediction based on hidden Markov models that not only outperforms the median prediction but also uses much less memory (as history is encoded in the model).

In the future we enhance our prediction to adapt the model at runtime to continuously changing plug behaviors.

Screencasts of our solution can be found at `www2.cs.fau.de/staff/mutschler/debs2014.html`.

Acknowledgements

We thank our students Michael Mroz and Cosmin-Inout Bercea for their help implementing the queries.

7. REFERENCES

[1] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 Requirements of Real-time Stream Processing," *ACM SIGMOD Rec.*, vol. 34, no. 4, pp. 42–47, 2005.

[2] H. Ziekow and Z. Jerzak, "The DEBS 2014 Grand Challenge," in *Proc. 8th Intl. Conf. Distributed Event-Based Systems*, (Mumbai, India), 2014.

[3] C. Mutschler, N. Witt, and M. Philippsen, "Demo: Do Event-Based Systems have a Passion for Sports?," in *Proc. 7th Intl. Conf. Distributed Event-Based Systems*, (Arlington, TX), pp. 331–332, 2013.

[4] G. M. Adelson-Velsky and E. M. Landis, "An Algorithm for the Organization of Information," *Soviet Math. Doklady*, vol. 3, no. 5, pp. 1259–1263, 1962.

[5] L. R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," in *Reading in speech recognition*, (San Francisco, CA), pp. 267–296, 1990.

[6] G. S. Manku, S. Rajagopalan, and B. G. Lindsay, "Approximate medians and other quantiles in one pass and with limited memory," *ACM SIGMOD Rec.*, vol. 27, no. 2, pp. 426–435, 1998.

[7] G. Beliakov, H. Bustince, and J. Fernandez, "On the median and its extensions," in *Comp. Intelligent Knowledge-based Systems Design*, pp. 435–444, 2010.

[8] F. Chin and H. Ting, "An improved algorithm for finding the median distributively," *Algorithmica*, vol. 2, no. 1-4, pp. 235–249, 1987.

[9] V. Sood, D. Fischer, J. Eklund, and T. Brown, "Developing a communication infrastructure for the smart grid," in *Conf. Electrical Power Energy Conference*, (Montreal, Canada), pp. 1–7, 2009.

[10] A. Papalexopoulos and T. Hesterberg, "A regression-based approach to short-term system load forecasting," *IEEE Trans. Power Systems*, vol. 5, no. 4, pp. 1535–1547, 1990.

[11] A. Bakirtzis, J. Theocharis, S. J. Kiartzis, and K. Satsios, "Short term load forecasting using fuzzy neural networks," *IEEE Trans. Power Systems*, vol. 10, no. 3, pp. 1518–1524, 1995.

[12] H. Hippert, C. Pedreira, and R. Souza, "Neural networks for short-term load forecasting: a review and evaluation," *IEEE Trans. Power Systems*, vol. 16, no. 1, pp. 44–55, 2001.

[13] A. Nizar and Z. Y. Dong, "Identification and detection of electricity customer behaviour irregularities," in *2009 Conf. Power Sys.*, (Seattle, WA), pp. 1–10, 2009.

[14] X. Li, C. Bowers, and T. Schnier, "Classification of energy consumption in buildings with outlier detection," *IEEE Trans. Industrial Electronics*, vol. 57, no. 11, pp. 3639–3644, 2010.

[15] S. Bu, F. Yu, P. Liu, and P. Zhang, "Distributed scheduling in smart grid communications with dynamic power demands and intermittent renewable energy resources," in *Conf. Communications Workshops*, (Tokyo, Japan), pp. 1–5, 2011.