DEBS Grand Challenge: Solving Manufacturing Equipment Monitoring Through Efficient Complex Event Processing

Tilmann Rabl[1,2,4], Kaiwen Zhang[1,3,4], Mohammad Sadoghi[1,3,4],

Navneet Kumar Pandey[5,6]*, Aakash Nigam[2,4], Chen Wang[2,4], Hans-Arno Jacobsen[1,2,4]

[1]Middleware Systems Research Group
[2]Department of Electrical and Computer Engineering
[3]Department of Computer Science
[4]University of Toronto

[5]Department of Informatics
[6]University of Oslo

ABSTRACT

In this paper, we present an efficient complex event processing system tailored toward monitoring a large-scale setup of manufacturing equipment. In particular, the key challenge in the equipment monitoring is to develop an event-based system for computing complex manufacturing queries coupled with event notifications and event and query result visualization components. Furthermore, we present an experimental evaluation to validate the effectiveness of the proposed solution with respect to both query latency and throughput.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Information filtering

General Terms

Experimentation,Performance,Measurement

Keywords

Event processing architecture, Publish/Subscribe, Complex Event Processing, event storage, DEBS Grand Challenge

1. THE ACM DEBS GRAND CHALLENGE

For a number of years, the ACM International Conference on Distributed Event-based Systems (DEBS) has identified a Grand Challenge for the event processing community. The 2012 DEBS Grand Challenge centers around processing monitoring data, in which the use case is targeted at monitoring high-tech manufacturing equipment [4]. In a nutshell, the challenge is to develop a (centralized) event-based system for computing complex manufacturing queries together with notifications and visualization components. To evaluate the wide applicability of the proposed approach, every solution must be able to support a set of complex continuous queries (provided by ACM DEBS) consisting of a specific set of monitoring stream operators targeted at the manufacturing use case. Furthermore, to test the effectiveness of the proposed solution, the posed queries must support a high event throughput rate (high-throughput) and meet stringent latency requirements (low-latency).

The monitoring data is periodically sent in form of Google Protocol Buffer (GBP) messages. The baseline frequency of these messages is 100 Hz. The data itself was recorded from an actual system and each reported message consists of 66 values. The values are a time stamp in nanoseconds and sensor readings in form of Boolean and numeric values. Out of the 65 recorded measurements 9 are subject to the event processing. The 9 measurements are reported from 3 different kinds of metrics (3 of each type):

- Electrical Power Main Phase (int)
- Chemical Additive Sense (Boolean)
- Chemical Additive Release Valve (Boolean)

The rest of the measurements are not used and can be discarded. The first metric measures the power consumption of the manufacturing equipment. The later two measure the state of a sensor for a chemical additive and the state of a release valve for the additive. As stated above three independent instances of each of these metrics are reported in every message.

Based on this information, two continuous queries (provided by ACM DEBS) are computed. The first query measures the time difference between the state change of the sensor and the valve. This difference is plotted over 24 hours and if there is a change of more than 1 percent within 24

*Work performed while visiting Middleware Systems Research Group at the University of Toronto.

DEBS '12, July 16–20, 2012, Berlin, Germany.

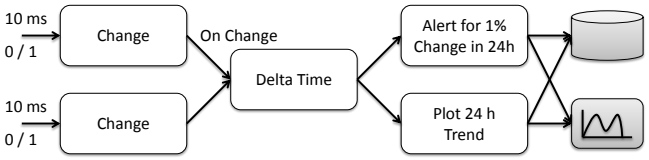

Figure 1: Challenge Query 1

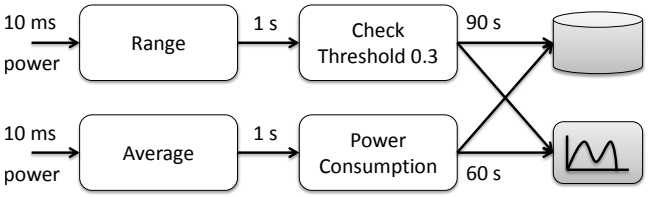

Figure 2: Challenge Query 2

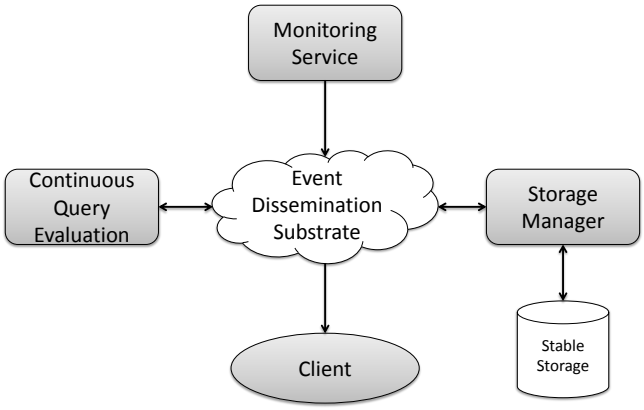

Figure 3: High-Level Architecture

hours an alarm has to be raised. The second query calculates the average and the relative variation of each power consumption metric per second. The average power consumption is stored for one minute. If there is a variation of 30 percent in one of the three power monitoring measurements, the raw power data has to be stored for at least 90 seconds (20 seconds before the threshold violation and 70 seconds after the threshold violation.) In the following section, a more detailed description of the queries is given. The results of the queries have to be persisted to disk and may be displayed using a graphical user interface.

For the experimental evaluation, two real data sets are provided by ACM DEBS: a small five minute sample and a large data set that contains 27 days worth of data. Both were recorded at an actual manufacturing unit. These can be replayed (and streamed) with a data generator that produces the GBP messages. To test for the maximum throughput, the frequency can be increased by the desired factor. The provided data features all violations that are queried. However, they are infrequent and, therefore, we implemented an additional synthetic data generator that produces more frequent violations. For example, in the real data consecutive changes in the sensor readings can have a time difference of up to 11 hours.

1.1 Manufacturing Queries

Two standing queries are defined in the challenge. The queries are continuously evaluated on the incoming event data stream. The first query monitors three sensors and the corresponding valves. An abstract representation of this query can be seen in Figure 1. The data arrival frequency is 100 Hz. Input to the query are 6 Boolean values where 3 present the sensors and 3 the valves[1]. Note that Figure 1 only shows one combination of sensor and valve. If the sensors report a certain measurement, the valves are switched accordingly. In the first part of the query, changes are registered and reported with a timestamp. The second part of the query correlates the change of the sensor and valve, and reports the time difference. This time difference is stored for 24 hours. Whenever the time difference increases by over 1% within the window, an alarm must be raised. Furthermore, the trend of the time difference within a 24 hour window is plotted using simple linear regression. The plot is updated whenever a new value is reported.

The second query operates on energy monitoring data. An abstract overview can be seen in Figure 2. Here, for 3 different measurements, averages and ranges are computed in one second windows. In the figure we, again, only present the processing of one of the 3 measurements. The first level operators compute the average and the range of values for a one second window. Since the data rate is 100 Hz, this aggregates approximately to 100 values[2]. If the range of values within the window exceeds a certain threshold, an alarm must be raised and the raw data (all energy measurements) has to be recorded in a window of 20 seconds before and 70 seconds after the violation. The average power consumption is calculated every 60 seconds and output for every measurement independently.

[1]Due to a problem in the data recording, the query has actually only 3 input values that are combined to 3 different pairs.

2. ARCHITECTURE

The high-level architecture of our solution to the challenge is comprised of five main components (see Figure 3). These components are considered logical and could be physically distributed. Our implementation is centralized and encapsulates the entire architecture in a single application binary, running on a single machine.

Monitoring Service: The monitoring service represents the input event data stream. In the challenge, it consists of the generation of raw data and its marshalling to the appropriate format (e.g., Google Protocol Buffers). In our implementation, the function of the monitoring service is by and large given through the challenge specification.

Event Dissemination Substrate: The event dissemination substrate is in charge of disseminating events (e.g., monitoring data points) between the different components. In a distributed setting, this substrate could be supported by a publish/subscribe system [6, 5, 3]. However, for simplicity, in our centralized setting, we use a queue manager which coordinates several producer and consumer queues. Data produced by the monitoring service is consumed by the query operators, which serve the results back to the queue man-

[2]The actual number can be slightly more or less since it is not guaranteed that data arrives exactly every 10 ms.

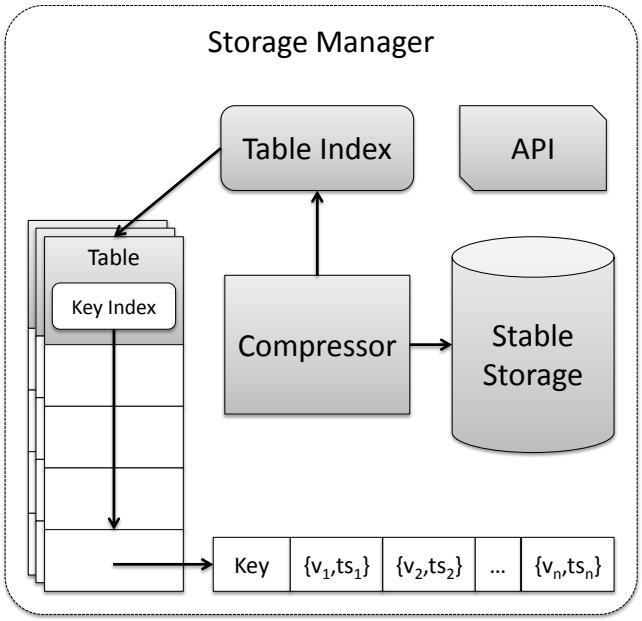

Figure 4: Storage Architecture

ager, which, in turn, makes the results available to clients and the storage manager.

Continuous Query Evaluator: The queries consume the monitoring data and produce new results by constantly evaluating the state of the query operators against the input stream. The output events from this component are either directed at the storage manager, or the clients.

Storage Manager: The storage manager is in charge of storing data. In addition, we leverage the storage manager to support window-based queries, which requires collecting data points over a time- or count-based window. In particular, the intermediate results of a sliding window computation are also stored and retrieved through this component.

Client: Finally, the client is able to visualize the results of the query. For instance, alerts and graph summaries are used to display relevant information to the user.

2.1 Storage Model

Query output and intermediate results are stored in our key-value storage module. Figure 4 highlights the architecture of our storage component. Data is partitioned in tables, which are created at run-time. The storage manager maintains an index for fast table lookups. Each table then contains specific key-value data, indexed by key. Each key maintains a linked list of values in increasing timestamp order. Finally, the compressor component takes a snapshot of the data as input and compresses the current state of the data held in-memory and sends it to stable storage (or transfers it across the network.)

In our storage model, new tables are created by calling the `create(table, schema)` function on the storage manager. The table schema (a comma-separated list of attributes) used is purely advisory; no checks are performed to validate insertions. Instead, a data value is stored as a single string. The schema is stored as table metadata and can be requested by clients to facilitate parsing of the data.

The storage module is designed for sustaining high insertion rates. To simplify the concurrency model, our storage supports only insertions (i.e., append only): a key-value pair is considered immutable once inserted. Each new value is inserted at the tail of a list maintained for each key. The list is ordered (i.e., versioned) by increasing timestamp order, which allows for efficient binary search. The ordering invariant is maintained by verifying that inserts happen in an increasing timestamp order; an insert which attempts to insert a value older than the latest inserted value is rejected.

Furthermore, querying is limited to only two types of primitives: `read()` and `scan()`.

- `read(table, key, value, timestamp)` returns the closest value to the timestamp (without exceeding it) for the given key. A null timestamp indicates a read request for the latest value.

- `scan(table, key, value, start, end)` returns a list of values that falls in the specified $[start, end]$ time interval for the given key.

Finally, data in our storage model is persisted on-disk by calling the *snapshot*() command. Once data has been compacted to disk, the memory is once again reallocated for the newly arriving data. For compacting data, the run-length encoding (RLE, c.f., e.g., [10, p. 20]) algorithm is used on a per-key basis to compress the data. Since the storage module is used to store output variables as separate keys, it is common for successive key values to have repeated values; hence, maximizing the effectiveness of RLE. It has to be noted that RLE is not used in the evaluation, though. This is due to fact that the data has to be stored in a raw form in the challenge. However, our tests with the provided data showed that very high compression rates would be possible since the average run length for the boolean values is 20000. This would allow size reduction as high as 99.99 percent.

2.2 Google Protocol Buffers

Most client-server platforms use a serialization technique to serialize into a leaner data format and then, de-serialize on the receiver's end. We use Google Protocol Buffers [1] as default serialization format in our system. GPBs are a flexible, efficient, automated mechanism for serializing structured data. They provide a smaller footprint, an improved processing speed, and a simpler programming interface compared to XML.

Specification of information to be serialized is defined using protocol buffer message types in .proto files. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs. After defining the messages, the protocol buffer compiler is run for the application language on the .proto file to generate the data access classes. These classes provide simple accessors for each field as well as methods to serialize and parse the whole structure to and from raw bytes. These classes can be generated for multiple different technologies, which means each class can be generated for both the client and server technologies. The protocol buffer technology provides the ability to update the data structure without breaking deployed programs that are compiled against an old format.

In our system, GPBs are widely used for storing and interchanging all structured information between various interfaces and modules. We define multiple different data structures and services in .proto files for GPB serialization for

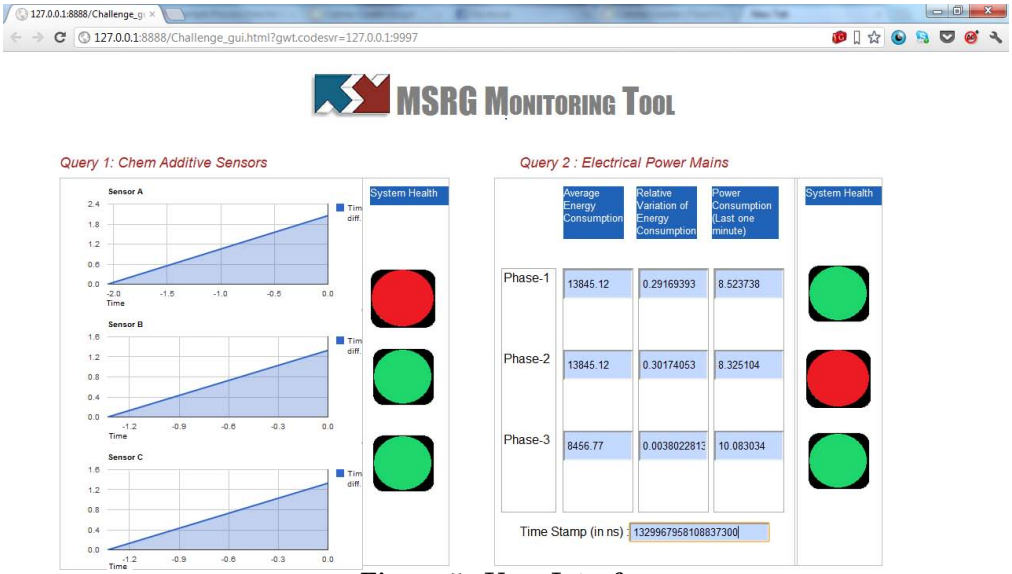

Figure 5: User Interface

different message types. The provided data generator outputs serialized events in GPB formats which are then processed by query operators. The data produced by queries are pushed into graphical interface for visualization and are subsequently stored through GPB serialization.

2.3 Client

We decided to design an Ajax-based user interface using Google Web Toolkit (GWT) [2] for our project. GWT is a set of open-source tools that allows Web developers to create and maintain complex JavaScript front-end applications in Java. Apart from necessary native libraries, it is completely implemented in Java and can be built on any supported platform with the included GWT Ant build files. GWT emphasizes reusable, efficient solutions to recurring Ajax challenges, namely asynchronous remote procedure calls, history management, bookmarking, internationalization and cross-browser portability. When the application is deployed, the GWT cross-compiler translates the Java application to standalone JavaScript files that are optionally obfuscated and deeply optimized.

Our GWT-based user interface consists of two root panels, one for Query 1 and another for Query 2. As show in Figure 5, the first panel visualizes the constant monitoring of the trend for the time difference using the least square method for a period of 24 hours for the chemical additive sensors. Whenever, the time difference increases by more than 1% within a 24 hour period an alarm is raised. The second panel, outputs average values and relative variations for each sensor every second, calculated over the period of one second. It also outputs power consumption of the manufacturing equipment in one minute intervals.

2.4 Implementation Details

Evaluation criteria for the challenge are (1) correctness, (2) throughput, and (3) latency![4]. Therefore, we optimized our code to improve the latter two criteria without compromising the first criterion.

The data generator, provided as part the challenge, produces the measurement data in GPB format and includes a simple server that can receive these messages. We built our server implementation on top of this server to ensure compatibility with the data generator. The two required queries are completely independent and were, therefore, implemented separately. To further reduce the amount of unnecessary work, we have merged the three measurements in each query into a single operator.

Conceptually, the first set of operators simply register changes between two successive measurements. The change is reported to an operator that calculates the delta times between the changes of a sensor reading and the related valve reading of the manufacturing equipment. Since these operations are simple, we implemented them as a single operator.

The Query 1 sliding window (a 24 hours time-based window) is maintained lazily. Whenever a new change event is received, all values that are older than 24 hours are discarded. The sliding window data are next fed into a linear least squares computation by leveraging the `SimpleRegression` class from the Apache Commons Mathematics Library [11]. The query has two kinds of output: plotted data and alarm notification (triggered when the threshold of 1% delta increase is violated.) Both outputs are written to separate files as serialized GPB messages. Query 1 only outputs data in cases of changes. These do not happen periodically in the test data. Therefore, for most events Query 1 has little computational overhead.

In Query 2, all data has to be recorded for some time. This means that in contrast to Query 1, Query 2 always induces an overhead upon receiving a new event. This overhead is partly attributed to the need for keeping an event history (up to 20 seconds) in case when the Query 2 threshold is violated. Therefore, we maintain a buffer that stores all events seen in the last 20 seconds.

Data is processed in one second chunks for computing the data range and average, therefore, it is advantageous to maintain the buffer in the same chunk size. Every second, the oldest one-second data chunk is discarded provided there was no violation. The total power consumption is stored for 60 seconds and output in a separate GBP file. The computation of the total power consumption is done based on the one second data chunk and stored in an array with 60 fields. In the case of a violation of the data range (i.e., range$>$ 0.3)

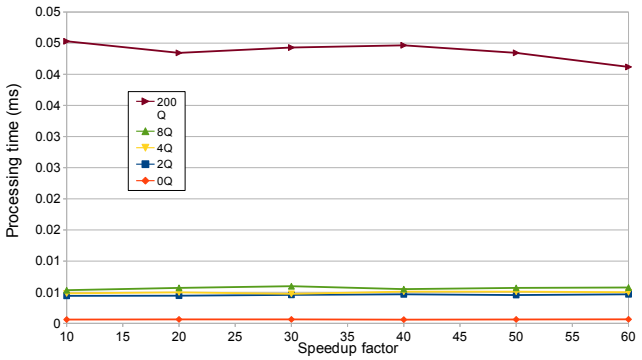

Figure 6: Evaluation of Processing Overhead

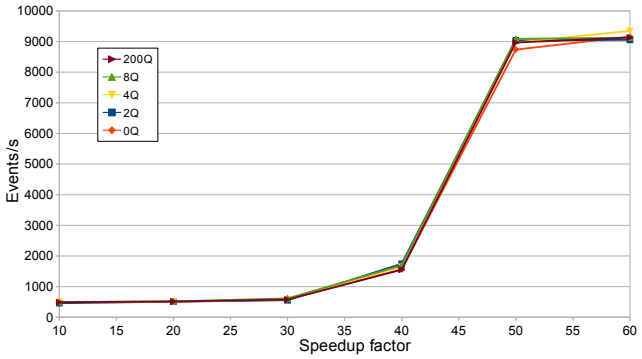

Figure 7: Evaluation of Throughput

an alert is stored in a GBP file and 20 seconds worth of raw data before the first violation and 70 seconds worth of raw data after the last violation (in the same violation sequence) are stored in a raw dump.

3. EVALUATION

We evaluated our system in the way that it would be deployed in a production system. Therefore, we used a distributed setup where the data generator resides on a separate machine from our system. All the experiments are conducted using two machines each with four 1.86 GHz Xeon processors and 4 GB of RAM. One machine runs the data generator, while the other machine runs the processing server via a socket connection. For the experiments we used the 5 minutes data set provided by the challenge. In order to achieve best results, we did not start the graphical user interface during the evaluation.

Metrics: We evaluate two metrics: system throughput and latency. Throughput refers to the rate of event processed per second. An event is considered processed only after it has been evaluated by all the queries in our system. Latency refers to the server processing time required to evaluate the event against all the queries.

Parameters: We perform our tests with varying speedup factor (at the data generator) and a varying number of queries at the server. For instance, the query workload 0Q means that there are no queries registered in our system, i.e., every event, upon receipt by the server, is simply discarded [3]. Similarly, the query workload 4Q means that there are four queries in our system. To obtain four queries, we duplicate the Query 1 and the Query 2 (cf. Section 1.1). Also, to achieve a fair comparison, we do not exploit the overlap among similar queries, and we consider duplicated queries as distinct. We also use two different workloads: the real sample provided by the challenge (REAL) and a synthetic one we have prepared (SYNTH). REAL does not contain any detectable conflicts for the queries and thus constitutes a best case scenario. SYNTH generates events at a constant rate of 100Hz. Conflicts for Query 1 every 65$\dot{0}$00 events (i.e., 10 minutes, 50 seconds) and conflicts for Query 2 at least every 30$\dot{0}$00 events (i.e. 5 minutes). We also ran a complete test with the full data set to test our system. However, because of the size, the resulting runtime behaviour is prohibitive for exhaustive testing.

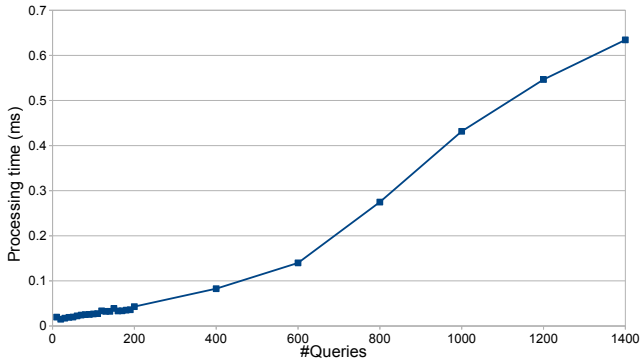

Figure 8: Synthetic Workload Performance Latency

3.1 Real Workload

Latency: Figure 6 shows that the throughput stays stable for increasing speedup factor. However, the latency does increase as the number of queries increase: for 200 queries, the processing time per event is still under 0.05 ms.

Throughput: We observe that between a speedup factor 40-50, the throughput is significantly increased while reaching its peak performance at 50 (see Figure 7). This is expected since the event rate is controlled by the speed at which they are produced by the data generator. However, the throughput saturates at around a factor of 50. Since the performance is the same across the different query workloads, we deduce that the bottleneck is the incoming data rate. At a rate of 9000 events/second, the server receives on average one event every 0.11 ms, which is well above the time required for even 200 queries. Thus, our server is actually waiting idle for a considerable amount of time as it is waiting for the next event to be processed. Possible causes for this bottleneck include the socket connection or other limitations in the data generator itself.

3.2 Synthetic Workload

Latency: The latency starts at 0.19ms up to 0.63ms for 700 queries (see Figure 3.2). Since the synthetic workload contains actual conflicts, the latency for this type of workload is expectedly higher than the real workload. However, it still scales relatively well to the number of queries.

[3]The query workload 0Q is intended for demonstrating the overhead of our system

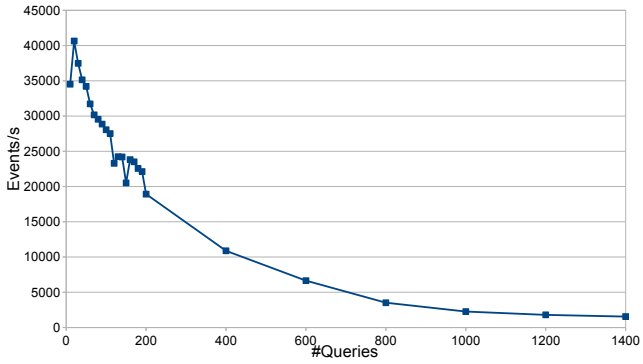

Figure 9: Synthetic Workload Performance Throughput

Throughput: We are capable of achieving a much higher throughput of 40000 events per second with this workload (see Figure 3.2). This is because the query evaluator is not constrained by the supplied data generator and its socket connection, which we suspect are the bottlenecks in the REAL workload. Beyond 700 queries, memory becomes an issue since an increasing amount of data has to be stored for each query.

4. OTHER USE CASES

In this section, we consider two other types of use case scenarios (of data sensor monitoring), which are conceptually similar to DEBS 2012 Grand Challenge problem and thus are target applications for the presented system.

Smart Traffic Monitoring With the proliferation of sensors in cars and the wide-scale adoption of GPS embedded in smart phones as a means to gather traffic information, sensor data is likely to play a critical role in future intelligent transportation systems. New sensor data generated by mobile devices and vehicles of many types (cars, truck, buses, rail transit) augmented with road infrastructure-based data will become available to support wide arrays of smart transportation applications. Such applications critically depend on reliable data services to collect, aggregate, and disseminate the available sensor data [8]. Applications for smart transportation infrastructures also depend on efficient event processing capabilities to discover events of interest in the continuous sensor data volume, to correlate events in order to identify emerging threats, and to filter out events in order to avoid overwhelming the infrastructure with needless information. In essence, our proposed complex event processing system tailored for manufacturing monitoring addresses major challenges in terms of the scale, granularity, and heterogeneity of the streamed sensor data; thus, our solution is readily extensible to other forms of sensor data monitoring use cases such as traffic monitoring.

Energy Monitoring Green computing is growing in interest focusing on designing environmental sustainability models that leverages energy-efficient computing with respect to both underlying hardware and software. To measure system compliance with green computing standards various metrics for assessing energy usage, tools and methodology for efficient power management, and energy monitoring applications have been proposed [7]. Systems such as e-surgeon [9] stress that energy efficient systems should monitor not only at the gross hardware-level but also within individual applications. To monitor these systems, a large amount of real-time information must be collected, analyzed, visualized; these goals are also conceptually similar to our manufacturing monitoring use case, thus, making our solution applicable here as well.

5. CONCLUSIONS

In this work, we tackled the ACM DEBS 2012 Grand Challenge problem of monitoring a large manufacturing equipment by developing a special-purpose complex event processing engine. We presented an architecture amendable to distributed execution that consists of five main components. (1) The monitoring service that provides a data transformation component to process incoming data streams in various data formats such as GPB. (2) The event dissemination substrate that provides underlying communication among various components in our system. (3) The continuous query evaluator that supports complex queries required by the manufacturing use case of the challenge. (4) The storage manager that consists of an efficient in-memory data store coupled with on-disk persistent storage enhance by data compression. (5) The client that provides a graphical interface for summarizing the current state of the manufacturing equipment. Finally, we demonstrate the effectiveness of our proposed solution through an extensive experimental evaluation of both query latency and throughput.

6. REFERENCES

[1] Google Protocol Buffers. http://code.google.com/p/protobuf.

[2] Google Web Toolkit. https://developers.google.com/web-toolkit.

[3] H.-A. Jacobsen, A. K. Y. Cheung, G. Li, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh. The PADRES Publish/Subscribe System. In *Principles and Applications of Distributed Event-Based Systems*. IGI Global'10.

[4] Z. Jerzak and K. Kadner. Debs 2012 grand challenge.

[5] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. ICDCS'05.

[6] G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. Middleware'05.

[7] S. Murugesan. Harnessing Green IT: Principles and Practices. *IT Professional'08*.

[8] T. Nadeem, S. Dashtinezhad, C. Liao, and L. Iftode. TrafficView: a scalable traffic monitoring system. In *MDM'04*.

[9] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. e-Surgeon: Diagnosing Energy Leaks of Application Servers. Technical report, INRIA Tech Report'12.

[10] D. Salomon. *Data Compression: The Complete Reference*. Springer, 3rd edition, 2000.

[11] The Apache Software Foundation. The Apache Commons Mathematics Library, 2004.