TechniBall: DEBS'2013 Grand Challenge

Avigdor Gal, Sarah Keren, Mor Sondak, Matthias Weidlich
Technion - Israel Institute of Technology
avigal@ie.technion.ac.il
{sarahn,mor,weidlich}@tx.technion.ac.il

Hendrik Blom, Christian Bockermann
TU Dortmund
hendrik.blom@tu-dortmund.de,
christian.bockermann@udo.edu

ABSTRACT
In this work we present the solution to the DEBS'2013 Grand Challenge, as crafted by the joint effort of teams from the Technion and TU Dortmund. The paper describes the architecture, details the queries and offers our observations regarding the appropriate way to trade-off high-level processing with time constraints.

Categories and Subject Descriptors
H.3.4 [**System and Software**]: Distributed Systems

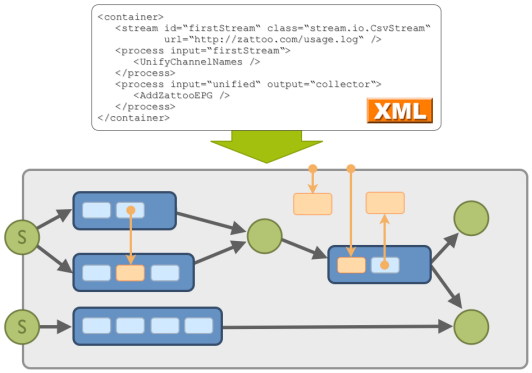
Keywords
event processing, real-time event processing

1. INTRODUCTION
The ACM DEBS Grand Challenge series seeks to provide a common ground and evaluation criteria for event-based solutions, by offering problems that require event-based systems and that can be evaluated using real-life data and queries. The 2013 challenge involves real-time complex analytics over high velocity sensor data using the example of analyzing a soccer game. The input data comes from sensors in the players shoes and a ball used during a soccer match. The real-time analytics involves continuous computation of four main statistics types, namely ball possession, shots on goal, heat maps, and running analysis of team members.

This paper presents our *TechniBall* solution that was developed at the Technion – Israel Institute of Technology together with the Technical University in Dortmund. The proposed solution strikes a balance between the use of high-level complex event processing language (Esper [3]) and a fast low-level processing of events that arrive at the pico-second level (*streams* [1]).

The paper is structured as follows: In Section 2 we detail the overall architecture and discuss implementation aspects. Section 3 presents the queries that support the required analytics. We share our experiences in building and running the system in Section 4 and Section 5 concludes the paper.

2. ARCHITECTURE
The TechniBall solution is based on the open-source *streams* framework [2], which provides a description language for data flow graphs consisting of sources, sinks and processors. The data flow is defined in high-level XML and is compiled into a computation graph for a stream processing engine (e.g. the *streams runtime* or a topology for the *Storm* engine [5]). *streams* aims at providing a clean and easy-to-use Java-based middleware to design and implement data stream processes. The *streams* library promotes simple software design patterns such as JavaBean conventions and dependency injection to allow for a quick setup of streaming processes using simple XML files. This leads to a rapid prototyping suite for designing and implementing streaming processes as data flow graphs. Besides the rapid prototyping, *streams* allows for direct integration of custom processor classes into the XML configuration, which allows for an easy adaption of the framework to specific use cases. For TechniBall we used the latest *streams* version 0.9.10-SNAPSHOT.



As part of the provided TechniBall solution, we integrated Esper [3] into the *streams* framework by implementing a single custom Esper processor using the *streams* API. Esper is a component for complex event processing (CEP), available as a Java framework. It enables rapid development of applications that process large volumes of incoming events in real-time. Esper filters and analyzes events using a declarative high-level, SQL-like language, responding to conditions of interest with minimal latency. We used Esper 4.6.0 for Java, open-source software available under the GNU General Public License (GPL). With the integration of Esper we are able to directly include Esper queries into the XML description of the data flow graphs using a simple `esper.Query` tag.

Figure 1 shows the definition of a computation graph with a single input stream and a process that includes the Esper process.

```
<container id="sample-config">
  <properties>
    <!-- define attribute types of attributes
        used in ESPER queries below -->
    <esper.types>...</esper.types>
  </properties>

  <!-- define a stream implemented by the custom soccer stream
       class that has been optimized for the DEBS data  -->
  <stream id="soccer" url="file:/path/to/game-data"
       class="stream.io.FastSoccerStream" />

  <!--  define a process node connected to the soccer stream
        that applies the esper engine and send results to the
        queue "R"                          -->
  <process input="soccer">
    <stream.esper.Query output="R" types="${esper.types}">
            Esper QUERY
    </stream.esper.Query>
  </process>

  <!--  define a process connected to "R" that simply prints
        out all items to stdout    -->
  <process input="R">
     <PrintData />
  </process>
</container>
```

Figure 1: A simple graph with a stream and process including an Esper query.

The final TechniBall implementation comes as a single executable Java archive that includes the *streams* runtime, the Esper engine and needs to be provided with the data flow graph definition in an XML file. The XML definition for the DEBS challenge has been crafted to distribute processing among multiple threads. To comply with the challenge specification we created a virtual machine image running Ubuntu Server Edition 12.10 (64bit, kernel 3.5.0). We tested this image with virtual machine that features the predefined 4 cores (@2.8 GHz) and 4 GB RAM.

Figure 2 provides an overall architecture of the TechniBall system. The *streams* framework provides stream and queue implementations (green circles) and the notion of processes (dark blue boxes), which read from these and execute a list of processors (light blue boxes) for each item read. The top left stream S represents the data input (e.g. file). Items read from the stream as simple hash-maps, which can contain arbitrary Serializable values. This allows for dynamically adding new fields to the data which can be consumed by subsequent processors.

A single reader process is used which reads in the data and applies a set of operations (processors) to each element of the stream. These elements provide pre-processing such as joining meta-information to each sensor measurement (player ID, sensor type). The final processor of the reader process distributes copies of the sensor measurements to several queues, each of which is processed by its own process. The modular concepts of the processors allows different layout of the data flow graph (i.e. to compute some queries in a single thread). The shaded optional processor `GameView` for example can be added to display a live-view of the game (see Figure 9 in the discussion Section 4).

3. CHALLENGE QUERIES

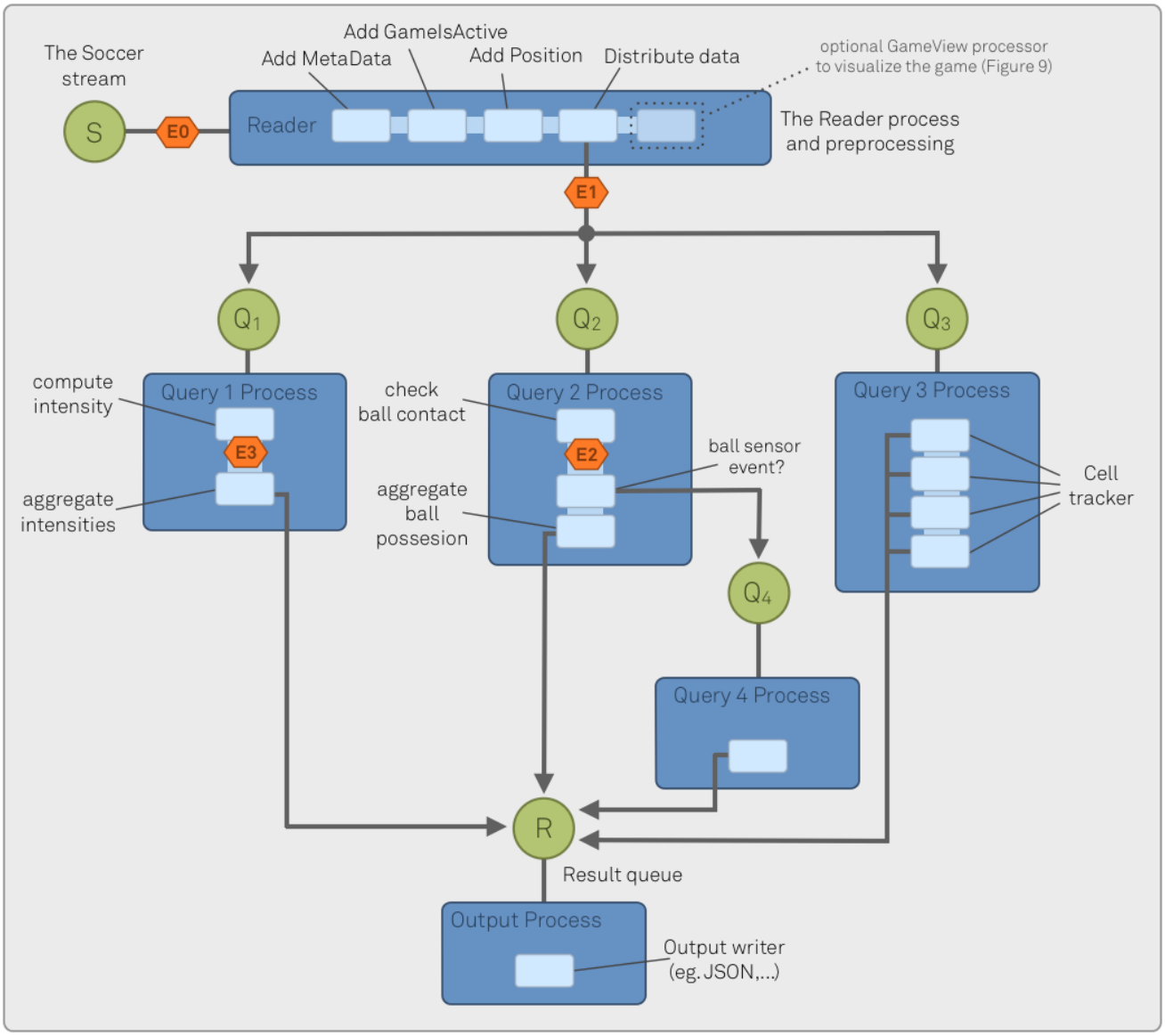
We now turn our attention to the four queries posed by the challenge. For most of the query implementations, the input data events need to be preprocessed. This pre-processing takes place by applying a chain of processors that have been implemented using the *streams* API. The data pre-processing enriches the data using a metadata file (in JSON format), which contains the player names, associates the transmitter IDs with a player ID (`pid`) and adds fields for the type of sensor (leg, arm, left, right). The player ID provided by the metadata is artificially set up and numbers the players from 0 to 17, where ID 0 is assigned to the referee sensors and all ball sensors are assigned a negative ID. In addition, all players of team A are assigned odd IDs and the players of team B are equipped with even IDs. This allows (a) for using arrays as data structures with the `pid` as the index to player data and (b) for using the *modulo* operator to determine which team the player belongs to. Other operators in that step add flags for signaling whether the game is currently active or compute the player from the leg sensors. The enriched events are marked as $E1$ in Figure 2.
After pre-processing, we added a final processor that distributes the enriched events into several queues (Q_1, Q_2 and Q_3) which are processed by separate threads each of which handles a different query. As the shot detection is part of query 2 and required in query 4, we add another queue Q_4, which is fed with the detected shots of the query 2 processing. Any processor within the query processes that sends query results is provided with a reference to a file result queue R. A final process simply reads from R and emits the items in R to standard output.

3.1 Query 1 – Running Analysis

Purpose of this query is to keep track of each players current running distance and additionally provide insight of the intensity of the runs (e.g. player trotting, player in sprint). The process for the running analysis is shown in Figure 3.
As a first step in the running analysis process, we implemented a processor that keeps track of the intensity level for each player. According to the challenge requirements, intensities need to stay stable over a full second before they are regarded as a new intensity state. Our `AddIntensity` processor implements this tracking of player intensities and emits the stable intensity levels for each player. For computing the aggregated intensities, we implemented two alternative approaches: First, we created an Esper query to aggregate the events directly using the SQL like Esper query language. Using the `esper.Query` processor as shown in Figure 1, we added the following Esper query for aggregation (only the trot level for a 1min time window, which is scaled to 60000 sec in Esper):

```
<esper.Query condition="%{data.pid} @gt 0"  output="R">
  SELECT pid AS player,
    sum(case when intensity=1 then distance else 0 end)
          as trot_distance,
    sum(case when intensity=1 then (ts_stop - ts_start)
          /1000000000 else 0 end) as trot_time
    FROM Data.win:time(60000 sec) GROUP BY pid
</esper.Query>
```

As an alternative solution, we directly implemented the aggregation as separate processor `AggregateCurrentIntensity` using the *streams* API. This custom processor exploits some



Event	Base Fields/Attributes	Additional Fields/Attributes
E0	id,ts,x,y,z,v,a,vx,vy,vz,ax,ay,az	
E1	all fields of e0	pid,active,pos.x,pos.y,pos.z,[player.x,player.y]
E2	all fields of e1	nearest,ball:hit,ball:hit:x,ball:hit:y, ball:hit:z,hit:leg:x,hit:leg:y,hit:leg:z,[ball:posession]
E3	all fields of e0	intensity

Figure 2: This figure outlines the data flow graph set up within the TechniBall system. The dark-blue boxes are mapped to parallel threads whereas the light-blue boxes are the processes that are being executed within those threads. The system data flow is defined within a single XML file. In the first step, each of the processors (light blue) was provided by the Esper engine integration. For more efficient processing, some of these queries have been replaced by custom implementations, e.g. for an efficient tracking of the heat maps (Section 3.3).
The table below shows the event fields at the various spots of the data flow.

of the features added in the pre-processing step such as direct array lookup of data structures using the `pid` value.

```
<process input="Q:1">
    <!--  Track current intensity for each player -->
    <stream.soccer.AddIntensity />

    <!-- Aggregate the intensity levels for each player
         and emit results to the queue "R"      -->
    <stream.soccer.AggregateCurrentIntensity output="R" />
</process>
```

Figure 3: The process definition for the running analysis.

3.2 Query 2 – Ball Possession

The ball possession query requires several parts of information to be gathered over all the events: The player current positions need to be handled and maintained; the ball position needs to be assessed; and the condition for a hit of the ball needs to constantly be checked. Finally the ball possession status needs to be maintained and the times of possession need to be aggregated according to the windows sizes. Figure 4 shows the XML definition of the process that handles the ball possession query.

```
<process input="Q:2">
    <!--  Add details on hits of the ball -->
    <stream.soccer.CheckBallContact />

    <!-- Handle possession changes and aggregate the
         possession times for the required windows    -->
    <stream.soccer.AggregateBallPossession output="R" />

    <!--  Send ball events for shot on goal detection -->
    <Enqueue condition="%{data.pid} @lt 0" queue="Q:4" />
</process>
```

Figure 4: The process for handling the ball possession query. Ball events (`pid` < 0) are forwarded to queue Q_4.

Our `CheckBallContact` processors keeps track of the current player positions and checks whether the *ball hit* condition is fulfilled for any incoming ball event. Keeping track of the players can again be done in constant time as the `pid` value from the pre-processing step serves as index to an array holding the position for a player. The *ball hit* condition is only checked for ball sensors which are within the field boundaries, thus, this processor automatically chooses the active ball for its computations.

If a hit of the ball is detected, the `ball:hit` field of the event is set to the `pid` of the player that hit the ball. In addition, this processor maintains the current possessor of the ball. Once the possessor changes, the possession time of the last player is emitted. If the ball leaves the field or the game is interrupted, possession is transferred to player with `pid` 0 (referee). The processor `AggregateBallPossessions` maintains sliding windows for each of the requested window sizes and aggregates the possession times provided by the previous processor. Upon changes of the possession, the current aggregations are emitted to the result queue. An alternative version for this aggregation can be performed by the Esper processor that we integrated into *streams* and the Esper query shown in Figure 5.

```
<process input="Q:2">
  <stream.soccer.CheckBallContact />

  <!-- Use Esper to aggregate the ball hits grouped
       by player over a sliding window     -->
  <stream.esper.Query condition="%{data.ball:hit} > 0" output="R">
    SELECT max(ts) AS ts, pid AS player,
      sum(‘ball:possession‘) AS time,
      sum(‘ball:hit‘) AS hits
    FROM Data GROUP BY pid OUTPUT LAST EVERY 1 EVENTS;
   </stream.esper.Query>
   <Enqueue condition="%{data.pid} @lt 0" queue="Q:4" />
</process>
```

Figure 5: Aggregation of ball possession grouped by player over a sliding window using the Esper query language.

3.3 Query 3 - Heat Map

We implemented the heat map tracking directly in a Java processor within the *streams* framework. Our `CellTracker` allows for specifying the granularity of the grid and pre-initializes the memory required to track all cells for each player. Each of the cell objects allocates memory for keeping the aggregation windows at initialization time as well. This is a direct memory-to-speed tradeoff – by pre-allocating the memory and using a customized data structure, we ensure updates of our heat-maps in $\mathcal{O}(1)$: With the player ID (pid) which we add in the pre-processing phase, the cell updates for the grid for each event is then possible in constant time. From the x and y coordinate of the sensor we can directly compute the index of the cell and the player ID is then used as the index of the counts that need to be updated. We include one instance of our `CellTracker` for each grid resolution as shown in Figure 6.
The output format of our `CellTracker` implementation is a little different from the format proposed in the original challenge description: instead of emitting an item for each cell (one line per cell), we provide a compressed form of the cell array as

$$cell\_times : [\ldots, i : v_i, (i+1) : v_{i+1}, \ldots] \; \forall \, i : v_i > 0.$$

Thus, we emit the index of the cell and its value for all entries larger than 0. The rest of the provided fields of the emitted result items for the heat map queries matches the format requested in the challenge description.

```
<process input="Q:3">
   <soccer.CellTracker gridx="8" gridy="13" output="R" />
   <soccer.CellTracker gridx="16" gridy="25" output="R" />
   <soccer.CellTracker gridx="32" gridy="50" output="R" />
   <soccer.CellTracker gridx="64" gridy="100" output="R" />
</process>
```

Figure 6: Definition of the process with the four cell tracker instances. Results are emitted to queue R.

3.4 Query 4 – Shot on Goal

The shot on goal query requires a shot/hit of the ball as a trigger and then checks if the ball is directed towards the opponents goal. The process for this query is attached to the process handling the ball possession query, as this query already emits information about detected hits.
The `ShotOnGoal` processor computes the future position of a ball based on such a *ball hit* event and checks of the future

position passes the goal areas as specified in the challenge description within the pre-defined amount of time. This future position is projected from the current position based on the direction and speed of the ball (also including gravity). Essentially, the respective processor can be in three states, *initial*, *hit detected*, and *shot-on-goal detected*. We transition from *initial* to *hit detected* if the `ball:hit` flag is set for a ball event. As part of that also the position of the leg that hit the ball is stored. As soon as the ball is more than a certain distance (the challenge allowed up to 1m) away from this position, we check whether it would reach the goal accordingly and, if so, transition from *hit detected* to *shot-on-goal detected*. Whenever being in this state, we emit the according events and check whether the conditions for terminating the shot-on-goal are satisfied. If so, we transition to state *initial*.

```
<process input="Q:4">
  <!--  Check for shot on goal, events require the
        ball:hit field and the gameIsActive flag -->
  <stream.soccer.ShotOnGoal output="R" />
</process>
```

Figure 7: The process handling query 4. The detection of shots (hits) is provided by annotations added by the `CheckBallContact` processor.

3.5 Query Result Formats

To meet the challenge requirements regarding the result output formats, we provide a single output process, which prints out the result items emitted by the different query implementations to standard output. As this leads to a mixture of result items from different queries being printed to the single standard output, a few additions have been made to allow for simple inspection and parsing:

1. We add a field `query` to each emitted line, which references the query for which this item has been emitted.
2. We encode each item as a JSON object and separate items by a line break.

The JSON encoding allows for additional fields that can be added. For example, for the heat maps, we emit items with the additional fields `query`, `grid` and `win`, which identify the query, the grid resolution and the window, to which the output line belongs.

```
<process input="R">
   <!-- Print to stdout in JSON format  -->
   <stream.io.JSONOutput />
</process>
```

Figure 8: The output process which includes a processor that dumps input to stdout in JSON formatted lines.

4. DISCUSSION

Having introduced the overall architecture and the specific implementations of the four queries, we now highlight some additional aspects of the implementation and consider future improvement.

The combined use of *streams* and Esper was done with an attempt to gain the best of both worlds. We have built upon the abilities of *streams* to perform fast processing of enrichment, filtering, and computation. As an example, *streams* provides interfaces to implement custom data sources. In the case of the soccer data we implemented a custom parser that directly creates values from each line without splitting the line into substrings. This avoids additional memory allocation and dramatically speeds up the parsing. Further possible improvements would be to read at the byte-level and read block-sizes that match the underlying hard-disk blocks. Esper, on the other hand, kicked in as soon as complex reasoning was needed, making use of its high level descriptive query language. With the integration of Esper as the generic `esper.Query` processor, we are able to directly apply Esper queries for the events processed by *streams*.

In general, the relative ratio of the *streams* code vs. the Esper queries depends on the system requirements. For some parts of the system, we have implemented both Esper queries and *streams* code, trying out the trade-offs of declarative application specifications, rapid programming, maintainability, execution timing requirements, *etc.* The heavy emphasis on throughput in the challenge led the implementation away from massive Esper coding and in the direction of more *streams* coding. Nevertheless, the Esper coding was not done in vain. It served as a basis for a deep understanding of the application needs. We see the search for optimal combination of a CEP engine with other tools as an important line of future research. As a concrete example, our analysis shows that much of the aggregation functions needed for the challenge rely on simple counting mechanisms. Hence, these aggregations are easy to implement directly in *streams* in order to achieve fast aggregation computation.

With the presence of four cores, our parallelization scheme allocated one core to the *streams* initial processing, one core to processing Query 1, one more core was allocated for processing Query 3 and the last core was allocated for processing queries 2 and 4, which share a significant portion of the computation. Moving to an environment with more processors, we intend to partition queries 1 and 3 to a by-player analysis by simply removing the `group-by` statement and adding a routing component in the distributer *streams* processor. For Query 3 processing can be further parallelized to the four separate cells.

The output of the system is currently sent to the standard console output, as required by the challenge. However, in a more advanced implementation, the final events are sent back to a *streams* processor, which visualizes the positioning of players and the ball, and visualizing heat maps as well. Figure 9 illustrates the position visualization, as created by the TechniBall system, with blue and red dots represent the players of the two teams and a white dot represents the position of the ball. It is worth noting that while the game was played on half a field, the visualization transforms it into a whole field setting. In the game visualization, the player who has possession of the ball is encircled. Also, as soon as a shot on goal is identified, the ball changes its color. In the image in Figure 9, the cyan colors show the detected ball hit, whereas the ball is still in possession of player 12 since no other player has yet hit the ball (even though player 10 is closer to the ball).

The grand challenge, as designed, tests correctness and throughput. We believe, however, that this dataset can serve as a benchmark testing for other aspects of event processing as well. For example, uncertainty management of sensor data and patterns is an intriguing topic that can be tested using this dataset (see a book chapter on the topic [4]). A task that involves probabilities of analysis such as the one presented in Query 4 (shot on goal) can use this dataset as a benchmark.

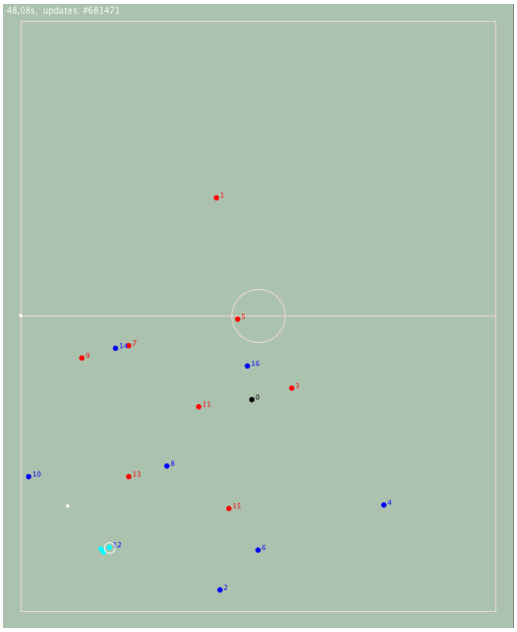


Figure 9: Visualization of the sensor measurements (live) by adding the `GameView` processor to the configuration file.

5. CONCLUSIONS

In this paper we have presented a solution to the DEBS 2013 grand challenge. The solution is based on parallelization and initial enrichments and filtering using *streams*, followed by high-level querying using multiple instantiations of Esper. Visualization of the output events is added, again, using *streams*.

The proposed solution is based on the framework developed for the INSIGHT European project.[1] The goal of the INSIGHT project is to radically advance our ability of copying with emergency situations in Smartcities by developing innovative technologies, methodologies and systems that will put new capabilities in the hands of disaster planners and city personnel to improve emergency planning and response The INSIGHT architecture requires a CEP engine that operates on top of a stream processing engine. Obviously, there are several choices both for CEP (*e.g.*, Esper, Event Calculus, *etc.*) and for stream processing (*e.g.*, *streams*, Storm, InfoSphere, *etc.*). Depending on the requirements of the application under consideration (*e.g.*, efficiency, verifiability, ability to deal with uncertainty, *etc.*), one makes the appropriate choice for the technology to adopt at each level. Here, *streams* provides a level of abstraction from the underlying event-processing system as it allows for mapping the data flow graph to different stream engines, such as the *streams runtime* or Storm.

An interesting aspect with the high-level description of the data flow graph is the partitioning of data and queries into several threads of execution. For the *TechniBall* setup we chose to divide this into three main parallel computation processes. However, with more CPUs or even additional compute nodes, this could be massively distributed on a larger set of processors. Finding the most efficient distribution and reasoning about strategies to generally find adequate partitionings of such compute graphs will be focus in future work.

[1] www.insight-ict.eu

Acknowledgement

We thank Ella Rabinovich for useful discussion. This research has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement number 318225.

References

[1] C. Bockermann. The *streams* framework, 2012. URL `http://www.jwall.org/streams/`.

[2] C. Bockermann and H. Blom. The streams framework. Technical Report 5, TU Dortmund University, 12 2012.

[3] EsperTech. Esper complex event processing engine, 2013. URL `http://esper.codehaus.org/`.

[4] A. Gal, S. Wasserkrug, and O. Etzion. Event processing over uncertain data. In S. Helmer, A. Poulovassilis, and F. Xhafa, editors, *Reasoning in Event-Based Distributed Systems*, pages 279–304. Springer, 2011.

[5] N. Marz. Storm - distributed and fault-tolerant realtime computation, 2013. URL `http://www.storm-project.net`.