Grand Challenge: The BlueBay Soccer Monitoring Engine

Hans-Arno Jacobsen[1], Kianoosh Mokhtarian[1], Tilmann Rabl[1], Mohammad Sadoghi[2], Reza Sherafat Kazemzadeh[1], Young Yoon[1], Kaiwen Zhang[1]

[1]Middleware Systems Research Group (MSRG), University of Toronto
[2]IBM T.J. Watson Research Center

ABSTRACT

This paper presents the design and implementation of a custom-built event processing engine called BlueBay developed for live monitoring of soccer games. We experimentally evaluated our system using a real workload and report on its performance. Our results indicate that BlueBay achieves a throughput of up to 790k events per second, therefore processing the game's input sensor stream about 60 times faster than real-time. In addition to our custom implementation, we also investigated the applicability of off-the-shelf general-purpose event processing engines to address the soccer monitoring problem. This effort resulted in two additional and fully functional implementations based on Esper and Storm.

1. INTRODUCTION

Complex Event Processing (CEP) systems in today's connected world define an exciting new area of research with rich potential applications and challenges. For the past two years, the ACM International Conference on Distributed Event-based Systems (DEBS) has organized annual Grand Challenge competitions aimed at promoting a common ground and common evaluation criteria for CEP applications. The DEBS 2013 Grand Challenge [9], which is the focus of this paper, considers the problem of event monitoring in a soccer game. This is an application scenario that is reminiscent of a wide variety of use cases that are made feasible as availability and accuracy of small wireless sensors increase and the ability of CEP systems for live processing of sensor data improves. Furthermore, while our events of interest in a soccer game are domain-specific and related to detection of various game conditions, they are at the same time representative of the challenges in the wider spectrum of application scenarios that involve continuous stream processing.

In this context, this paper presents multiple approaches to address the DEBS 2013 Grand Challenge. Our first solution is an event processing system called *BlueBay* that we designed and built from scratch for monitoring of soccer games. We discuss BlueBay's modular design that enables easy plugging of new types of soccer analysis queries and report on its performance in terms of throughput and delay, as well as the flexibility in tuning the trade-off between the two. Our results indicate that BlueBay achieves a throughput of 350k events/sec with a 90-percentile per-event delay of 0.005 ms. Alternatively, BlueBay can achieve a higher throughput of 790k events/sec with a 99-percentile delay of 15 ms.

In addition to BlueBay, we also investigated the use of existing open-source off-the-shelf CEP engines to solve the Grand Challenge problem. Our efforts in this regard resulted in two additional fully functional soccer monitoring implementations based upon Esper [2] and Storm [7]. We discuss our experience in the process of developing these solutions in qualitative terms and compare the relative performance of our implementations in quantitative terms.

Section 2 presents our multi-stage event processing pipeline that is intended to provide a unified framework under which our CEP solutions execute. Section 3 elaborates on different approaches that we considered in order to address the Grand Challenge monitoring problem. This includes Esper and Storm which resulted in working implementations, as well as STREAM and StreamIT that we were unable to use for the Grand Challenge problem. Section 4 presents the main contribution of this paper, namely the BlueBay engine and Section 5 reports on our experimental evaluation results.

2. MULTI-STAGE MONITORING PIPELINE

Figure 1 illustrates our soccer game monitoring pipeline consisting of three stages, namely, *(i)* the *sensor data collection and dispatching* stage, *(ii)* the *processing* stage, and *(iii)* the *visualization and distribution* stage. We briefly discuss each stage.

2.1 Stage 1: Data Collection and Dispatching

The first stage in our monitoring pipeline involves sensor data collection and dispatching. The input sensor stream originates from transmitters attached to the ball, the referee's and players' feet, as well as the goal keepers' feet and arms. A sensor reading contains the sensor's unique identifier `sid`, its `x`, `y` and `z` space coordinate, its `vx`, `vy` and `vz` velocity vector components, its `ax`, `ay` and `az` acceleration vector components, velocity vector magnitude `|V|`, and acceleration vector magnitude `|A|`. Each sensor's stream has a frequency of 200 events per second for the feet and arm transmitters, and 2000 events per second for the ball transmitter.

As shown in Figure 1, sensor readings collected from the soccer field are fed into our monitoring pipeline either directly or indirectly. A direct feed is suitable for online monitoring of a live game. Alternatively, sensor data streams can be timestamped and logged into a file (currently in CSV format), and fed into the pipeline at a later point. To support the offline processing mode, we developed a *network data dispatcher* that reads a game's sensor data log file and dispatches sensor readings over a socket connection.

2.2 Stage 2: Continuous Query Processing

The DEBS 2013 Grand Challenge outlines four continuous mon-

DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.

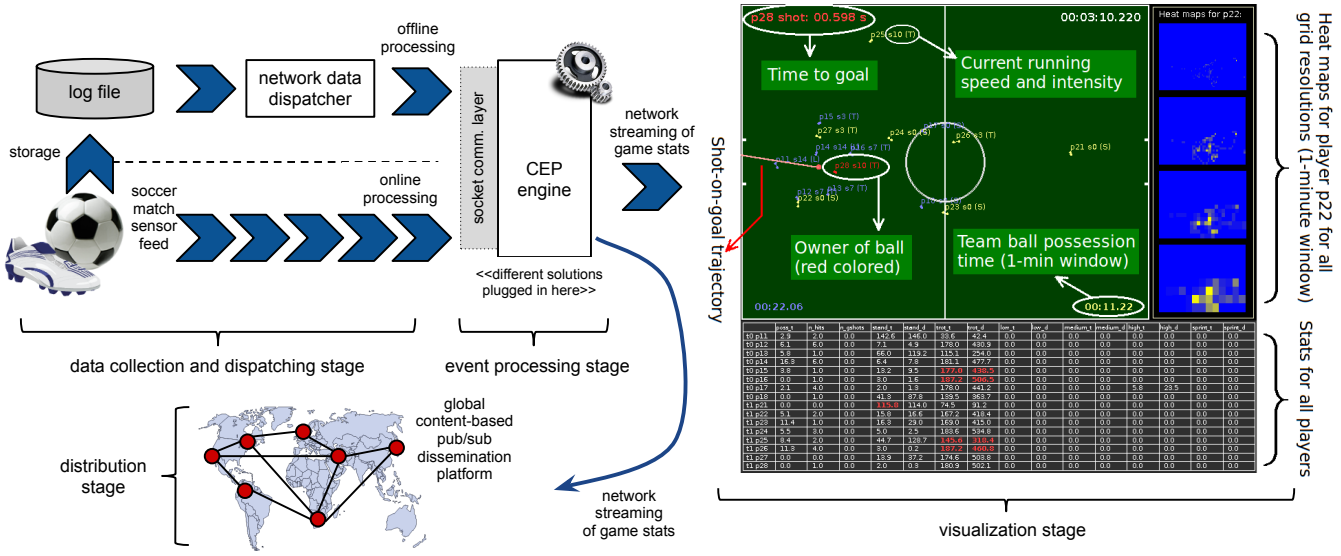

Figure 1: Unified soccer match monitoring pipeline.

itoring queries to detect different game conditions (*e.g.*, shots on goal) and gather various game statistics (*e.g.*, ball possession). The queries are executed in the second stage of our pipeline using different CEP engines that we developed for this purpose. In what follows, we give a brief overview of these monitoring queries (a more elaborate description can be found here [9]). We defer the discussion of different CEP engines to execute these queries to Section 3.

Query 1 (Q1) – Players' running performance: Q1 concerns monitoring of the players' running activity during the game based on 6 intensity classes (*i.e.*, *stop*, *trot*, *low*, *medium*, *high* and *sprint*). Players transition between these classes according to the momentary value of their speed. Q1 produces two types of outputs: *(i)* The intensity statistics output is produced at a maximum frequency of 50 Hz, and *(ii)* the aggregate intensity statistics output is produced based on four different time windows.

Query 2 (Q2) – Ball possession statistics: Q2 computes the time that the ball is in possession of a player or team. The criteria that must be satisfied for a player to possess the ball is to hit it such that his foot is within 1 meter of the ball and as a result of the hit the ball's acceleration reaches at least 55 m/s^2. Q2 produces outputs in two forms: *(i)* The per-player ball possession output reports the number of ball hits and the length of time each player possessed the ball, and *(ii)* the per-team ball possession output reports the percentage of time that the ball was in possession of each team.

Query 3 (Q3) – Heat map: Q3 produces statistics of the aggregate time that each player spends in different regions of the soccer field (*a.k.a.*, the *heat map*). For this purpose, four grid structures with different cell sizes are defined dividing the field from just about a hundred cells (in the least granular grid) to several thousand cells (in the most granular one). Q3 outputs the percentage of time that each player spends in each cell over different time windows.

Query 4 (Q4) – Shots on goal: Q4 detects players' shots on goal and produces an output stream that identifies the shooter and tracks the ball's motion towards the targeted goal. The necessary conditions that constitute a shot on goal require that the ball is hit by a player (similar to Q2), that it remains within the field, and that the extrapolation of the ball's trajectory indicates that it would be located within the goal posts' coordinates no later than 1.5 seconds after it was hit by the player, provided that the ball is not diverted.

2.3 Stage 3: Visualization and Distribution

The final stage in the pipeline is to provide the results of the query processing stage to the end-users. We envision two usage scenarios for these results. The first scenario involves the use of a graphical user interface to visualize the movement of the ball and players in real-time and to display various statistics in sync with the game play. We developed such a graphical monitoring panel that can be used by team coaches or TV reporters to analyze a live game (see Section 2.3.1 for more detail). Alternatively, the CEP engine's output can be multicast over the Internet for soccer enthusiasts. We believe that a content-based pub/sub network [4, 5] is a suitable choice for this purpose. Next, we discuss these alternatives in turn.

2.3.1 *The Graphical Monitoring Panel*

We have developed a GUI-based monitoring panel to visualize the soccer game and the related analysis information. In addition to a user friendly illustration of the information, this is a necessary tool for verifying the correctness of the different types of analysis (i.e., false positives and negatives). In our experience, given the huge volume of generated events, this verification would not have been feasible without our visualization tool (Figure 1 illustrates a screen shot and a sample video is available at [8]). We have designed this tool such that it displays the result of important event processing steps, a few examples of which include:

- The detected running speed and intensity is shown next to each player. Besides verifying the speed values, this helped us find significant fluctuations in the detected intensities in small (20 ms) intervals, if proper smoothing was not applied on the raw data.
- Every time a ball hit by a player is detected, the ball turns red and gradually goes back to its original color (white). This helped reveal false positive and false negative detection of ball hits.
- The tool highlights the player identified as the ball possessor. A timer tracks the reported ball possession time for each team (for the 1-minute window). This feature has contributed a lot to the high accuracy of our ball possession analyzer.
- Heat maps at different grid resolutions are displayed for a selected player (for the 1-minute window). Highlighted areas of the map track the player in the field, while his older positions gradually fade away as they fall outside the 1-minute window.
- Every time a shot-on-goal is detected, the ball's predicted trajectory and time to reach the goal is displayed and continuously updated until we leave the shot-on-goal state. Moreover, for every ball hit that is identified as a shot-on-goal, we visualize the ball's estimated next-1-second trajectory. This feature has helped us to tune and verify the accuracy of our shot-on-goal detector.

- A table of statistics displays for each player the time and distance spent in each running intensity level, ball possession time, and the number of ball hits. The table is continuously updated with recently changed values highlighted for easy tracking.
- The tool enables tracking of missed sensor events, *e.g.*, when no sensor data is received from a player's foot for 500+ ms. This feature is necessary to identify that the root cause of some missed events (such as shots on goal) are due to incomplete sensor data.

2.3.2 Pub/Sub-Based Dissemination Network

For the dissemination of query results, we envision the use of a distributed content-based pub/sub system [4, 5]. Soccer fans use the content-based filtering capabilities of these systems to tune into the statistics related to specific players or track various game conditions. For example, `subscription=[player:'Ronaldo', condition:'ball-hits']` encodes a user's interest to track Ronaldo's ball hits. As illustrated in Figure 1, the output of the query execution stage is fed into the pub/sub network during the game. This stream is matched against subscriptions at a broker and flows towards the users based on their subscription interests.

3. USE OF EXISTING CEP ENGINES

We now discuss the results of our investigation in the application of different off-the-shelf CEP engines to solve the Grand Challenge monitoring problems (see Section 2.2). We found that while some engines were sufficiently capable to address our query processing needs, others were not suitable for our purposes.

3.1 Esper

We used Esper open-source edition [2] (version 4.9.0) and successfully implemented all four challenge queries. The Esper distribution comes with a CEP engine and an event processing language (EPL) which provides a powerful interface into the vast capabilities of the CEP engine. The approach we took in our implementation is based on a logical decomposition of each of the challenge queries. At a high-level, this involves three phases: We first *pre-process* the raw sensor stream in order to augment it with game metadata information such as sensor type (*i.e.*, ball, referee, foot or hand sensor), player id, and team association (metadata is supplied as part of the challenge [9]). Next, in the *processing* phase, we formulate the challenge queries as smaller sub-queries that are continuously evaluated on the augmented stream and incrementally compute the final outcome (Figure 2 illustrates formulation of Q2 in EPL). Finally, the *reporting* phase uses time-triggered sub-queries to produce a sampled output of query results at designated frequencies.

We observed that Esper EPL features elaborate constructs (*e.g.*, windows, contextual partitioning, aggregation, expressions) that are directly applicable towards our stream processing needs. The Esper framework also supports seamless integration with the Java language at two key levels. First, it facilitates interfacing with the CEP engine for the input and output of events to and from the engine. Second, it allows incorporation of Java code snippets and functions as part of the evaluation of an EPL query. We used the former capability to inject raw sensor readings from our network data dispatcher and collect the computed output results using a Java callback method. We used the latter capability to encode the game-specific domain knowledge as stateless static Java functions, *e.g.*, to compute the ball trajectory (Figure 3 lists other Java functions).

Finally, we would like to reflect on our experience in working with Esper. We found that Esper's high-level language gives great flexibility and is suitable for fast development and ease of change.

```
insert into preprocessed_stream
select *,
  msrg.GameSetting.getId(s_id) as id,
  msrg.GameSetting.getType(s_id) as t_id,
  msrg.GameSetting.getSubtype(s_id) as subt_id
from msrg.EsperSensorEvent

insert into b_position_stream
select * from preprocessed_stream where t_id = 2

insert into b_relative_pos
select
  b.s_id as b_s_id, b.ts as b_ts, b.id as b_id,
  b.t_id as b_t_id, b.subt_id as b_subt_id,
  b.x as b_x, b.y as b_y, b.ax as b_ax, b.ay as b_ay,
  b.v as b_v, b.a as b_a, p.ts as p_ts, p.s_id as s_id,
  p.id as p_id, p.t_id as p_t_id, p.subt_id as p_subt_id,
  p.x as p_x, p.y as p_y, p.z as p_z,
  java.lang.Math.sqrt((p.x-b.x)*(p.x-b.x)+
    (p.y-b.y, 2)*(p.y-b.y, 2)) as dist
from preprocessed_stream as p unidirectional,
  b_position_stream.std:unique(s_id) as b
where (p.t_id = 3 or p.t_id = 4) and (b.t_id = 2)

create expression minDist
{(select min(dist) from b_relative_pos.std:unique(p_id))}

insert into b_possession
select *,
  msrg.GameSetting.getBallOwner(p_t_id,b_v,b_x,b_y) as owner
from b_relative_pos
where msrg.GameSetting.ballIn(b_x, b_y) = 1 and b_a > 0.5
  and minDist() = dist and dist <= 1000

insert into b_possession_percent
select *,
  sum(b_ts - prev(b_ts, 1)) as time_total,
  sum((b_ts - prev(b_ts, 1))
    * msrg.GameSetting.equalStr(owner,prev(owner,1),'teamA'))
      as time_teamA,
  sum((b_ts - prev(b_ts, 1))
    * msrg.GameSetting.equalStr(owner,prev(owner,1),'teamB'))
      as time_teamB
from b_possession.win:time(10 seconds)

select *,
  b_ts, owner, time_total,
  time_teamA/time_total as teamA_ownership_percent,
  time_teamB/time_total as teamB_ownership_percent
from pattern [every timer:interval(1 second)] unidirectional,
  b_possession_percent.std:unique(owner)
```

Figure 2: Query 2 implementation in Esper.

In fact, after a steep learning curve to become familiar with its capabilities, we were able to formulate all queries in a matter of a few days time. We therefore believe that in a use case with changing requirements an Esper-based solution is of great value. Also, as an added advantage of Esper's support of the Java language our solution can be incorporated easily into other programs.

3.2 Storm

Storm is a distributed stream processing system built for real-time Web scale stream processing [7]. A Storm *topology* is a directed acyclic graph that consist of data sources (*spout*) as roots and data processing nodes (*bolts*) as inner nodes and leaves. Spouts emit tuples that are consumed and processed by bolts. Spouts and bolts can have multiple instances that run in parallel. The way data is streamed to these instances is specified by *stream groupings*.

The most basic stream grouping is a *shuffle grouping* that randomly distributes data to bolt instances. A more advanced grouping is the *field grouping* that sends tuples with equal values in a given field to the same bolt. A Storm topology can be specified in various programming languages, for a more detailed discussion see [6].

`getId(s_id)`	Returns unique id for players, referee & ball wearing sensor `s_id`.
`getType(s_id)`	Returns type of entity (*i.e.*, team of players, referee & ball) wearing sensor `s_id`.
`getSubtype(s_id)`	Returns id for showing where sensor `s_id` is worn (*i.e.*, left/right leg/arm).
`equalStr(a, b, c)`	Returns `1` if `a=b=c` and `0` otherwise.
`ballIn(x, y)`	Returns `1` if `x, y` is a coordinate within field boundaries.
`getBallOwner(t_id, v, x, y)`	Returns `'ballout'` if `(x, y)` is within field or `'stopped'` if `v!=0`. Otherwise, returns `'teamA'` or `'teamB'` based on `t_id`.

Figure 3: Java functions from `msrg.GameSetting` used in Figure 2.

Figure 4: Storm topology excerpt for Query 3

Although spouts and bolts must be individually implemented, Storm takes care of the distribution of tasks (spout or bolt instances) and the reliable transmission of tuples between the two. A Storm system consists of a master node that distributes tasks, code and worker nodes that execute subsets of a topology. A Storm cluster is backed by a Zookeeper cluster [3], which keeps all state information and makes the cluster reliable and failure resilient.

We implemented all four queries in Storm using Java. Figure 4 shows part of the topology that computes Q3 (heat maps). Since all data comes from the sensor stream, there is only a single spout. As a first step, the sensor readings have to be matched with the players. This can be done with an arbitrary degree of parallelism using a shuffle grouping. Next, the sensor readings from the two feet of each player have to be joined to a single position. This is done with a field grouping and a maximum degree of parallelism in terms of the number of players. From the stream of each player's position, the individual heat maps are computed and sent to the output bolt which sends the results to the GUI or to standard output.

3.3 StreamIT

StreamIT [10] provides a high-level language with stream manipulation primitives. We unsuccessfully attempted to implement the challenge queries using StreamIT. We now list the limitations of StreamIT which prevented us from completing the queries:

Limited I/O support: StreamIT has restricted I/O capabilities.

Lack of powerful stateful operators: Communication between blocks in a StreamIT pipeline is regulated through different queues. Therefore, each block is essentially stateless and accesses additional data by peeking at various queues. This queue-based approach is not suitable to formulate the finite-state machine model required for our monitoring queries (such as Q2 and Q4).

Lack of library support: We believe that it would have been more beneficial if StreamIT was provided as a library, where the code produced by StreamIT could be modified in its lower level form.

3.4 STREAM

STREAM is a data stream management system to evaluate continuous queries developed by Stanford [1]. The main features of STREAM are to support a declarative continuous query language (CQL) that unifies access to an incoming (structured) data stream and traditional stored data (in form of tables).

CQL is formulated over either streams (an unbounded bag of events) or relations (a finite time-varying bag of events). CQL is also extended with a sliding time- and count-based window semantics, essentially, the sliding window is a snapshot of an observed finite portion of the event stream. There are three classes of operators in CQL. These classes of operators are distinguished based on their input/output semantics: (1) Relation-to-relation, which takes relations as input and produces a relation as output; (2) stream-to-relation, which takes streams as input and produces a relation as output; and (3) relation-to-stream, which takes relations as input and produces a stream as output. Another notable feature of CQL is the partitioning operator that partitions a stream into a set of sub-streams based on the values of selected attributes in the event stream. Despite the novel features of STREAM, the lack of direct support for user-defined functions (needed to implement complex finite-state machine behavior required by the DEBS Challenge queries) prevent us from including STREAM in our evaluation.

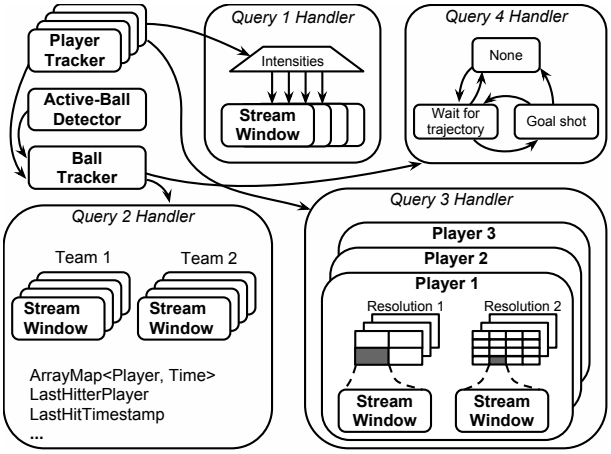

Figure 5: High level diagram of the BlueBay engine.

4. THE BLUEBAY ENGINE

BlueBay is the event processing engine that we have developed for analyzing a soccer game's sensor data. This engine is designed in a modular fashion to serve as a general framework for adding any type of soccer analysis query fairly easily (some discussed below), while it is also optimized for speed: With an efficient C++ implementation, it achieves a throughput of up to 790k events per second, handling the sensor events of a whole minute in only one second, *i.e.*, 60 times faster than real time.

BlueBay Components. The internal components of the BlueBay engine are illustrated in Figure 5 and briefly described in the following. The *Active-Ball Tracker* tracks which of the ball sensors is the one that we should be monitoring; only one of the balls is the one being played with. There are periods of a few seconds where there are temporarily more than one ball in the field, during which the Active-Ball Tracker should not be misguided until the extra ball is removed. Once the relevant ball sensor is identified, a *Ball Tracker* tracks the state of the ball—position, speed and acceleration, denoised with proper filters (exponential-weighted moving average). The Ball Tracker also monitors the acceleration of the ball and detects potential ball hits. We found that the acceleration is greatly impacted when the ball hits the ground, thus we take only the XY component of the acceleration into account (based on a threshold of 75 m/s^2). BlueBay also includes a *Player Tracker* for individual players which tracks the position and speed of the foot sensors. The speed is denoised with a sliding window moving average filter to cut out considerable fluctuations in the speed value as the players' feet takes every step. A value of {running speed, duration} is output every time the speed value is steady enough, with an output frequency of no more than 50 Hz and no less than 5 Hz. Upon every potential ball hit, the hitting player is identified based on the position of the foot sensors. Then, the *Trajectory Estimator* predicts the trajectory of the ball in the next few seconds. The Trajectory Estimator takes into account the current (denoised) position and speed of the ball as well as gravity and air resistance.

Stream Window. A common data structure used in the different components of BlueBay is a sliding window which keeps timestamped data values—entries of the form {timestamp, value, duration}—such as a 1-minute window on how much time a player has spent in a grid cell or the time a player has run at an intensity level. The key feature of this window, to which we refer as *Stream Window*, is its efficiency: It performs all insertion and retrieval operations in constant time and, more importantly, consumes only constant memory. This is achieved by bucketizing the timestamps at some granularity, and maintaining a list (in practice a circular

buffer) of fixed length, irrespective of the input rate. For example, to track the distance run by a player at the "sprint" intensity level in the past 5 minutes, we need to keep track of the (timestamped) distance values computed on each Player Tracker report (at 5 to 50 Hz). Instead of maintaining a sliding list of all values having a timestamp of no older than 5 minutes, we aggregate all values having a timestamp between T ms and T+999 ms in one bucket, and only maintain their sum and count, *i.e.*, 300 buckets. Compared to maintaining a full list of individual values, we do not lose any noticeable precision by such bucketization; in the worst case when we are pushing some outdated events out of the window, they may be leaving up to 1 second late. More formally, by maintaining the events of a sliding window of length T seconds in a Stream Window with N buckets, we may return the desired value (sum, max, count for the window) for the past T to $T(1+1/N)$ seconds, rather than exactly T. We use small buckets throughout BlueBay (*e.g.*, $N > 100$), except for the substantial number of heat map Stream Windows where we use $N = 20$; we report the heat map of the past 60 to 63 seconds for the 1 minute window.

Query analyzers. Given the above components, we conduct different types of analysis fairly easily. As illustrated in Figure 5, Q1 of the challenge can be readily handled by pushing the output of the aforementioned Player Tracker into Stream Windows—a Stream Window for each defined running intensity level. Similarly for Q2, we receive every ball hit along with the hitting player id from the Ball Tracker, and we can track the per-player and per-team possession time with a simple counter and a Stream Window (for each given window length), respectively. The heat map analyzer (Q3) is nothing but a collection of Stream Windows tracking data received from the Player Trackers. These windows, of different lengths and for the different grid resolutions, a total of 34,000 windows, easily fit into a few hundred MB of memory and perform all operations in constant time. Finally, the shot-on-goal analyzer (Q4) only needs to wait for ball hits from the Ball Tracker, and compare the trajectory given by the Ball Tracker with the position of the goals. This analyzer consists of a finite-state machine. Every time the ball is hit, we enter a waiting state for up to a maximum distance (50 cm) or a maximum time (1 second) since the hit. At that point, we have a reasonable estimate of the ball's trajectory and the time to hit the goal. Accordingly, we either enter the shot-on-goal state or go back to the no-shot state. In the former case, the trajectory is monitored upon every ball sensor event until the ball changes direction, becomes too slow, or leaves the field.

Other types of analysis. The components introduced above enable various new types of soccer analysis, such as the following.

- Ball passes and their success rate for each player and each team can be counted. We are signaled when a new player receives the ball (upon his first hit), and we know who has hit the ball last. We have already implemented this additional analysis in BlueBay.
- A player's running statistics with and without the ball, and at the time of attack and defense can be analyzed using a counter/Stream Window on the Player Tracker's output.
- Offsides can be detected by waiting for ball hits from the Ball Tracker, and forming a black list of players who are not allowed to receive that pass based on their positions at the time of hit (accessible via Player Trackers).
- The success of a player for man-marking his assigned opponent player can be analyzed through a Stream Window to which we periodically push the distance of the two players. These windows can tell us, for instance, that defender X was within a 2-meter distance of the assigned attacker 90% of the time.
- The defense-to-attack time for a team can be tracked by monitoring the Ball and Player Trackers, and maintaining a single time counter and a flag representing when, which team last acquired the ball (similar to the way the Q2 analyzer monitors the Ball and Player Trackers). An event is emitted once the ball enters the opponent's defense zone.

Efficiency. Since almost all operations in this challenge can be done in $O(1)$ time, it is a matter of efficient implementation techniques that enables a throughput boost in query processing. A few of these techniques employed by BlueBay are as follows. First, for different components we avoid using hash maps to track per-sensor data (even though lookups are $O(1)$). Instead, we assign an *index* $\in [0, \textit{Num Entities} - 1]$ to each entity, such as a sensor or a player. The index for each sensor/player is looked up only once per event in an id-to-index hash map. Then, all the remaining entity lookups, such as updating the position of a player, is done using array-based maps. Moreover, we note that the main body of the analysis consists of a substantial amount of mathematical operations. We avoid the unnecessary use of floating point operations where int64 precision would suffice, while carefully handling cases where there is a risk of overflow (*e.g.*, multiplying two values of picoseconds since epoch). We use efficient, array-based circular buffers instead of linked lists or elastic vectors wherever the data size is fixed, which is often the case—Stream Windows. Finally, the data dispatcher (see Section 2.1) that executes as a separate process and loads the entire data in advance eliminates the impact of disk I/O in the critical path of event processing.

Parallelization. BlueBay can run in single and multi-threaded modes. For the latter, we note that the events emitted to different output streams, per query and sometimes per sub-query, should not contain *reordered timestamps*. Thus, we do not run an arbitrary-size thread pool to handle the events. Rather, we run one thread per query, with the ball possession and shot-on-goal analyzer combined in one since they include common steps. Our code profiling analysis has shown that the most time consuming event processing step in BlueBay is the *emission of heat map statistics*—over half a million Stream Windows that should emit data. Even just iterating over all the windows for all players takes up to a few dozen milliseconds. We therefore designate a number of sub-threads in the thread handling the heat map query, which are responsible for the different grid resolutions. Reordering of output events across different grid resolutions is fine since they belong to different output streams. Once the emission timer fires, the heat map thread launches these sub-threads and blocks until they are all done, before handling any new heat map input event. We also note that in all query analyzers, we disable emission to stdout or file by default, since it involves significant I/O that will not let us analyze the actual performance of the event processing engine. Note that, however, we do perform all the string formatting and preparation of the final output; we just do not print it (*i.e.*, `sprintf` to memory buffers rather than `printf` to stdout).

5. EVALUATION

We experimentally evaluated the performance of the BlueBay system in terms of throughput and per-event delay—the time it takes from when an event is received from the data dispatcher to when processing is finished and a possible output event is emitted. Our evaluations are conducted on a PC workstation with Intel Xeon 3.20 GHz 4-core CPU with 6 GB of memory. We measure the throughput as the number of events processed per seconds (e/s), and as a *speedup*. A speedup of s indicates that s seconds worth of sensor data are processed in 1 second worth of actual processing time (*i.e.*, events processed s times faster than real-time).

BlueBay running all four challenge queries in non-threaded mode achieves an average throughput of 364k e/s, and an average speedup

Implementation	Q1	Q2	Q3	Q4
BlueBay	141x	165x	30x	187x
Esper	7.5x	2.4x	6.3x	2.3x
Storm	9.7x	8.6x	9.8x	8.6x

Table 1: Event processing speedup using different approaches.

of 27.6x. Table 1 compares the per-query speedup of different implementations and Figure 6 plots the instantaneous throughput for BlueBay. By enabling the multi-threaded mode in BlueBay, the performance is on one hand increased and on the other hand impacted by some noticeable overheads (discussed below). The overall throughput can be increased to 790k e/s (2+ times higher).

As described earlier, in the threaded mode, we allow a query of type X to be processed while a query of type Y is taking some time. We can enforce the different query threads to run at more or less the same pace by limiting the input queue size of the threads. A queue size of 1 enforces that no thread can start working on a new event (it is not given one) until another thread finishes the previous event. A queue size of ∞ makes the threads completely independent. The queue size also governs a trade-off between throughput and per-event delay: A limited queue size ensures that when a thread is falling behind (typically the event triggering heat map emissions), the other threads are paused after some point, giving the full CPU to the busy thread (sub-threads handling heat map emissions; see Section 4). Figure 6 illustrates BlueBay's performance in multi-threaded mode with different queue sizes (q).

Note that unlike I/O-involved jobs where threading immediately boosts the performance by avoiding wasting CPU cycles, here, all threads carry out CPU-heavy jobs, so for small queue sizes the overhead of threading is more significant than the obtained throughput increase in Figure 6. A major part of this overhead is that of the safe enqueuing/dequeuing of events for the threads which takes a non-negligible time compared to the only-a-few-microsecond processing time for most events. Another, less significant factor is the slight performance drop by turning on additional monitoring features such as collecting delay information and tracking the 99-percentile delay; turning on these features in the non-threaded mode drops the aforementioned throughput of 364k to 346k e/s. Finally, unlike I/O-involved jobs, here, the overhead of the many context switches between the CPU-intensive threads is non-negligible.

The trade-off between throughput and delay is illustrated in Figure 7. We can observe in this figure that the throughput can be increased only up to some point, after which some threads (the heat map thread) have a hard time catching up; hence, the jump in the delay value. In particular, the average/99-percentile processing delay for the last three queue sizes, 4k, 16k and 64k is 2.9/15 ms, 13/61 ms and 56/253 ms, respectively. The average throughput for these cases is 747k, 789k and 790k e/s. This pattern which can be

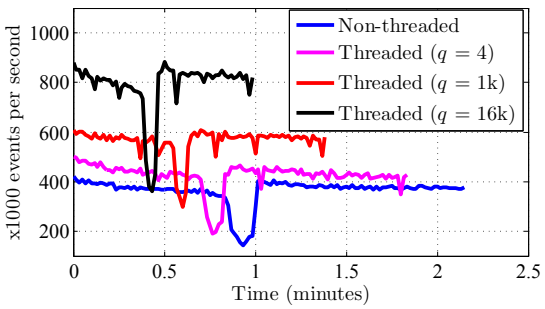

Figure 6: BlueBay's throughput (best viewed in color): x-axis is processing time and higher throughput executions end earlier (drops in graphs correspond to moments of missing ball data).

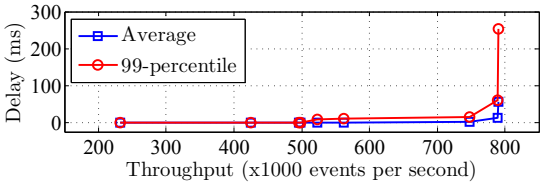

Figure 7: Trade-off between throughput and per-event delay. Points from left to right represent experiments with thread queue sizes of 1, 4, 16, 256, 1k, 4k, 16k and 64k, respectively.

seen in Figure 7 shows the maximum throughput. For non-threaded executions (not shown in the figure), the average/90-percentile delay and throughput are 0.004/0.005 ms and 346k e/s, respectively.

It is noteworthy that the rather complex relationship between throughput, delay and queue size is mainly due to: (i) the substantial unevenness between the work taken for the different events—in particular the *emission* of heat maps (even only preparing the output without the final push to stdout/file), and to a lesser extent, the running statistics—and (ii) the enforcement of one thread per query (and one per sub-query for heat maps) to avoid reordering of timestamps in the output streams, described in Section 4.

Summary. Among the scenarios described, we recommend to use BlueBay in non-threaded mode for monitoring of live soccer games. In this mode, BlueBay processes sensor events 27+ times faster than real-time with less than a few microseconds of delay for 99% of events (maximum delay is 75 ms for heat map emission). For offline monitoring of pre-recorded games, however, a slightly higher delay is not an issue and the multi-threaded mode with a queue size of 4k (or 16k) is recommended. This provides a throughput of 747k e/s (/789k e/s), a speedup of 57x (/60x) and a 99-percentile delay of 15 ms (/61 ms).

6. CONCLUSION

We presented the design of the BlueBay event processing engine that serves as a framework for conducting various types of analysis over the sensor data stream of a soccer game. BlueBay provides the flexibility to trade off between processing throughput in favor of per-event processing latency (and vice versa). Our measurements carried out using a commodity PC demonstrate that it can achieve a throughput of up to 790k events/sec. Moreover, we investigated the applicability of several existing CEP engines to address the Grand Challenge problem. This effort resulted in two additional implementations based upon Esper [2] and Storm [7]. Finally, we reported on our experience in the development process.

7. REFERENCES

[1] A. Arasu et al. Stream: the stanford stream data manager (demonstration description). In *SIGMOD'03*.

[2] Esper Tech Inc. The Esper complex event processing platform.

[3] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC'10*.

[4] 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.

[5] R. S. Kazemzadeh and H.-A. Jacobsen. Opportunistic multipath forwarding in content-based publish/subscribe overlays. In *Middleware*, 2012.

[6] J. Leibiusky, G. Eisbruch, and D. Simonassi. *Getting Started with Storm*. O'Reilly Media'12.

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

[8] Middleware Systems Research Group. Soccer game monitoring – sample video, 2013. `http://msrg.org/datasets/blue-bay`.

[9] C. Mutschler, H. Ziekow, and Z. Jerzak. The DEBS 2013 Grand Challenge. In *DEBS 2013: Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, Arlington, TX, USA, July 2013. ACM.

[10] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*.