Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, 37212

Real-time Sensor Data Analysis Processing of a Soccer Game Using OMG DDS Publish/Subscribe Middleware

Kyoungho An, Sumant Tambe, Andrea Sorbini, Sheeladitya Mukherjee, Javier Povedano-Molina, Michael Walker, Nirjhar Vermani, Aniruddha Gokhale, Paul Pazandak

TECHNICAL REPORT

ISIS-13-102

June, 2013

Real-time Sensor Data Analysis Processing of a Soccer Game Using OMG DDS Publish/Subscribe Middleware

Kyoungho An[1], Sumant Tambe[2], Andrea Sorbini[2], Sheeladitya Mukherjee[1], Javier Povedano-Molina[3], Michael Walker[1], Nirjhar Vermani[1], Aniruddha Gokhale[1], and Paul Pazandak[2]

[1] Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN 37212, USA
[2] Real-Time Innovations
Sunnyvale, CA 94089, USA
[3] CITIC-UGR
18071 Granada, Spain

Abstract. This paper describes a real-time event-based system to distribute and analyze high-velocity sensor data collected from a soccer game case study used in the DEBS 2013 Grand Challenge. Our approach uses the OMG Data Distribution Service (DDS) for data dissemination and we combine it with algorithms to provide the necessary real-time analytics. We implemented the system using the Real-Time Innovations (RTI) Connext^{TM}DDS implementation, which provides a novel platform for Quality-of-Service (QoS)-aware distribution of data and real-time event processing. We evaluated latency and update rates of one of the queries in our solution to show the scalability and benefits of configurable QoS provided by DDS.

1 DEBS Grand Challenge Problem

The ACM International Conference on Distributed Event-based Systems (DEBS) Grand Challenge problem comprises real-life data and queries in event-based systems. The goal of the DEBS 2013 Grand Challenge is to implement an event-based system for real-time, complex event processing of high velocity sensor data collected from a soccer game [1]. The real-time analytics for the DEBS Grand Challenge comprises continuous statistics gained by processing raw sensor data such as running analysis, ball possession, heat map, and shot on goal. Contemporary approaches to gathering data for soccer matches utilizes a complex system of 16 cameras to record every part of a soccer field and computers to process the feed from the cameras. The DEBS Grand Challenge adopts a different approach by using inertial sensors to collect all data.

The sensor data is collected by a real-time localization system from an actual soccer game. The data is recorded in a file and provided to the DEBS Grand Challenge teams. The sensors are located near each player's shoe, in the ball, and attached to each goal keeper's hands. The sensors attached to players generate data at 200Hz while the ball sensor outputs data at 2,000Hz. Each data sample contains the sensor ID, a timestamp in picoseconds, and three-dimensional coordinates of location, velocity, and acceleration.

The challenge problem requires that every team must address the four different queries it includes. Query 1 has two sub-queries: *current running statistics* and *aggregate running statistics.* The streamed result of current running statistics should include distance, speed, and intensity of each player. The intensity describes the running state of a player and is classified into 6 states (stop, trot, low, medium, high, and sprint) depending on the speed of that player. The aggregate running statistics are calculated and streamed per player over four different time windows: 1 minute, 5 minutes, 20 minutes, and the whole game duration. Multiple metrics are collected in these statistics.

The aim of query 2 is to calculate the ball possession time for each player and for each team. We calculate the total time of the ball possession and the number of hits for each players. For team ball possession, the total time of the ball possession and percentage of ball possession for a specified team are generated. Like query 1, the team ball possession statistics must be provided for four different time windows. Finally the statistics for these queries need to be updated with at 50Hz.

Query 3 provides heat map statistics capturing how long each player stays in various regions of the field. The soccer field is divided into defined grids with x rows and y columns (8x13, 16x25, 32x50, 64x100) and results should be generated for each grid size. Moreover, distinct calculations need to be provided for different time windows like query 1 and query 2. As a result, query 3 must output 16 result streams, a combination of 4 different grid sizes and 4 time windows, in total. Each stream includes the percentage of time that a player spends in each user-defined cell boundary.

Query 4 provides statistics for a shots on goal, which is defined as a player attempting to score a goal. It needs to keep track of shots scored or saved by the goalkeeper or other players. The result stream of the query includes the player ID and the ball sensor data.

The rest of the paper describes our solution to the challenge problem. Section 2 describes the details of our solution; Section 3 presents experimental evaluations of our implementation; and Section 4 presents concluding remarks including lessons learned.

2 Our Solution Using the OMG DDS-based Query Implementation

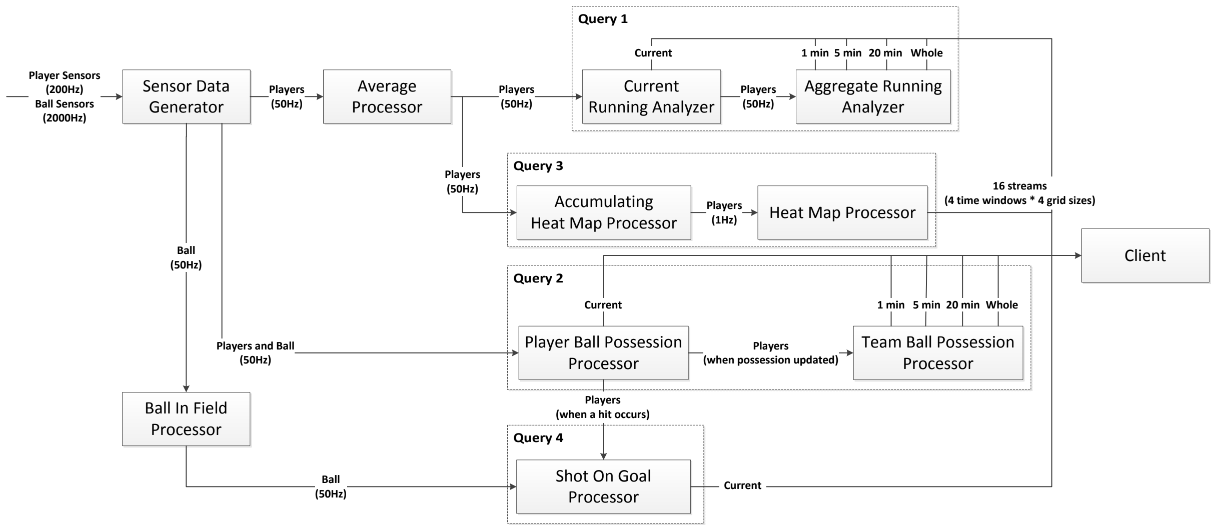


Fig. 1. High Level Data Flow Architecture

This section describes our OMG DDS-based solution to the DEBS 2013 Challenge Problem. We provide an overview of DDS and an overview of the solution with the DDS features used in our system. Finally, we present a detailed description of each query implementation.

2.1 Overview of OMG DDS

The OMG Data Distribution Service (DDS) specification defines a distributed communications middleware standard that is useful in a wide variety of environments [2]. The core DDS specification defines a data-centric publish-subscribe architecture for connecting anonymous information providers with information consumers. DDS promotes loose coupling between system components. The information consumers and providers are decoupled with respect to time (*i.e.,* they may not be present at the same time), space (*i.e.,* they may be anywhere), flow (*i.e.,* information providers must offer equivalent or better quality-of-service (QoS) than required by the consumers), behavior (*i.e.,* business logic independent), platforms, and programming languages. A data provider publishes typed data-flows, identified by names called Topics. The coupling is expressed only in terms of topic name, data type schema, and the required and offered QoS attributes of consumers and producers respectively. In the following subsection, DDS concepts used in our solution are briefly introduced.

Data-Centric Architecture Using DDS, applications share a *global data space* (or *domain*) governed by schema specified using the XTypes [3] standard. Each topic schema is described as a structured datatype (an Interface Definition Language *struct*). The datatype can be keyed on one or more fields. Each key identifies an instance (similar to a primary key in a database table) and DDS provides mechanisms to control the lifecycle of instances. Instance lifecycle supports CRUD (create, read, update, delete) operations. Complex delivery models can be associated with data flows by simply configuring the topic QoS.

DataWriter and DataReader *DataWriters* and *DataReaders* are end-points applications used to write and read typed data messages (samples) from the global data space. DDS ensures that the end-points are compatible with respect to the topic name, datatype, and the QoS. Creating a DataReader with a known topic and datatype implicitly crates a *subscription*, which may or may not match with a DataWriter depending upon the QoS.

Data Caching Data received by the DataReader is stored in a local cache. The application accesses this data using one of two methods: *read()* and *take()*; *read()* leaves the data in middleware cache until it is removed by either calling *take()* or it is overwritten by the subsequent data samples. The *Resource Limits* QoS prevents the middleware caches from growing out of bounds. Finally, *query conditions* provide a powerful mechanism to write SQL-like expressions on the datatype members and retrieve data samples that satisfy the predicates.

Content-Filtered Topics Content-filters refine topic subscription and filter samples that do not match the defined application-specified predicate. The predicate is a string encoded SQL-like expression based on the fields of the datatype. The query expression and the parameters may change at run-time. Data filtering may be performed by the DataWriter or DataReader.

DDS Quality-of-Service DDS supports multiple QoS policies. Most QoS policies have requested/offered semantics and can configure the dataflow between each pair of DataReader and DataWriter. We used the following QoS policies in our solution.

- **Reliability**: controls the reliability of the data flow between DataWriters and DataReaders. BEST_EFFORT and RELIABLE are two possible alternatives. BEST_EFFORT reliability does not use any cpu/memory resource to ensure delivery of samples. RELIABLE, on the other hand, uses an ack/nack based protocol to provide a spectrum of reliability guarantees from *strict* (*i.e.,* fully reliable) to BEST_EFFORT. The reliability can be tuned using the History QoS policy.
- **History**: This QoS policy specifies how much data must be stored (in DDS caches) by a DataWriter or DataReader. It controls whether DDS should deliver only the most recent value (*i.e.,* history-depth=1), attempt to deliver all intermediate values (*i.e.,* history depth=*unlimited*), or anything in between.
- **Time-based Filter**: High rate data flows can be throttled using the time-based filter QoS policy. A *minimum separation* configuration parameter specifies the minimum period between two successive arrivals of data samples. When configured, the DataReader receives only a subset of data.
- **Resource Limits**: This QoS policy controls the memory used to cache samples. Maximum number of samples, maximum number of samples-per-instance and maximum number of instances can be specified.

2.2 DDS-based Architecture

Our implementation of the four queries uses a DDS-based solution. The key architectural pieces of our solution, illustrated in Figure 1, comprise processing blocks and a high-level data flow. Each processing block consists of one or more DDS DataWriters and DataReaders to publish or subscribe to data, respectively, where the data flow in the network is logically partitioned by Topics. Using DDS, application developers do not need to consider the network connections of each data flow between DataWriters and DataReaders because it is handled by the underlying middleware. DataReaders and DataWriters and their QoS configurations in our system are defined in XML files that make it convenient to manage them and separate implementation concerns of the middleware and business logic of processing blocks. The architectural setup is geared towards solving all four queries of the DEBS challenge problem.

We defined data structures in IDL for each dataflow as well as its point-to-point segments. Key attributes, such as *sensor_id* and *player_id*, define unique *instances* to group the updates for the same physical entity (*e.g.,* a sensor). Through the grouping mechanisms enabled by defining a key in DDS, a processor can easily distinguish between multiple players. The keys also help in partitioning the dataflow(s) across multiple processors to exploit inherent parallelism in the *per-player* data streams. Each DataReader in the processing blocks has a (in-memory) cache to store and access the samples when needed. For instance, we utilized the cache data to implement the historical statistics required in some queries, such as aggregated data for the specified number of minutes. Cached data was also used for calculating distances between the ball and players to find the closest player to the ball for query 2 at a certain time.

In our solution, DDS QoS has been utilized for obtaining benefits of efficient implementation as well as satisfying the real-time requirements of the DEBS Challenge Problem. Since the queries have different characteristics in terms of the required update rates and processing data sizes, we needed to apply the appropriate QoS configurations for each query according to its characteristics. We set BEST_EFFORT as *Reliability* QoS for periodic queries like query 1 and 3, and RELIABILITY for aperiodic queries like query 2 and 4. Consequently, our system can guarantee faster update rates for periodic queries while assuring accuracy for aperiodic queries. Additionally, we utilized *Time-based Filter* QoS to manage resources efficiently and guarantee a query's requirement by configuring different minimum separation values. The details on how each query is implemented is presented in the rest of this section.

2.3 Implementing Query 1

The *SensorDataGenerator* processing block publishes sensor data in a real-time manner. The published data is sent to multiple processing blocks to be processed in parallel. In the case of query 1, a processor named *AverageProcessor* merges the data of two sensors attached to a player by averaging the sensor values. In the processing block, ball data is filtered because only the player's information is needed for query 1. We utilized *Time-based Filter* QoS supported by DDS to efficiently manage constrained resources and follow the required update interval of 50Hz. The next processing block for Query 1 is *CurrentRunningAnalyzer* in which a player's running state, speed and distance are calculated, and the results are published to clients and the *AggregateRunningAnalyzer*. Finally, in the *AggregateRunningAnalyzer*, the accumulated values of distance and speed categorized by a player's running states are streamed as a result, which is divided into multiple streams for the given time windows.

When a new sample arrives at the DataReader of a processor, a callback function, *on_data_available()* provided by the DataReader's listener interface is invoked. Since the processor of the query computes required statistics when a new sample arrives, we implement our business logic in the call back function.

By using x and y coordinates of a previous sample kept in a processor, distance is calculated. The timestamp of the previous sample is designated as the starting time of the calculation and the timestamp of the current sample is assigned as the ending time. The speed is obtained by the velocity value of the current sample and the unit of speed needs to be converted from μm/s to km/h.

However, the raw sensor data obtained from a real soccer game is not always accurate. Erroneous distance or speed values are possibly attained based on incorrect sensor data. To avoid the problem, we compare three speed values at the current timestamp obtained using three different ways: velocity of the current sample, velocity of the previous sample, and distance and time of the current sample. If outliers exist, these are removed by a filtering process.

Figure 2 presents distance in millimeters between each timestamp of a player and shows improvement of data accuracy by applying the filtering process. If differential between the speed values determined by different sources of sensor data at the same timestamp is larger than 20 km/h, which is configurable for different physical context, it can be marked as a sensor data error and needs to be fixed to a proper value from a correct source. Once speed and distance values are accurately evaluated, a player's intensity is classified by the speed value. If all of the required information is successfully calculated and stored, we call the write operation to let the middleware publish the outcome for clients.

For aggregate running analysis, we maintain a separate processor called *AggregateRunningAnalyzer*, which subscribes to the result stream of the current running analysis. The aim of the query for aggregate running

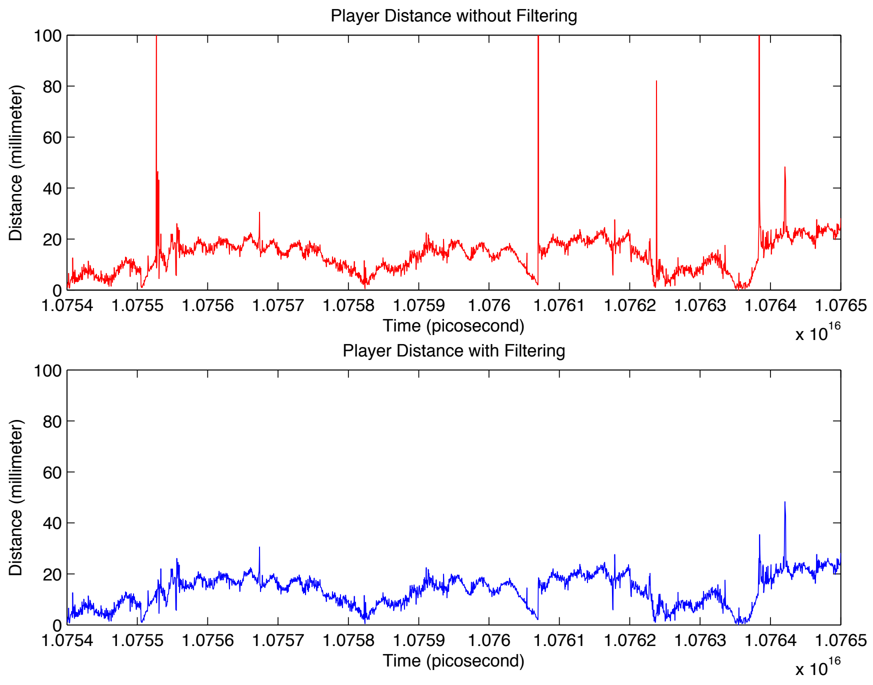


Fig. 2. Comparing Player Distance with Filtering Outliers

analysis is to accumulate the time duration of classified intensities of players for 4 different time windows. Since the query incorporates the four time windows, the processor publishes four result streams by different data writers.

The main challenge of this query is managing historical sample data effectively. We applied an incremental algorithm which avoids redundant computation that will occur when a new sample arrives. The algorithm keeps adding duration values of a new sample to a cache in the processor, and subtracting the time duration values of samples from the cache which are no longer included in the specified time windows because of newly arrived samples.

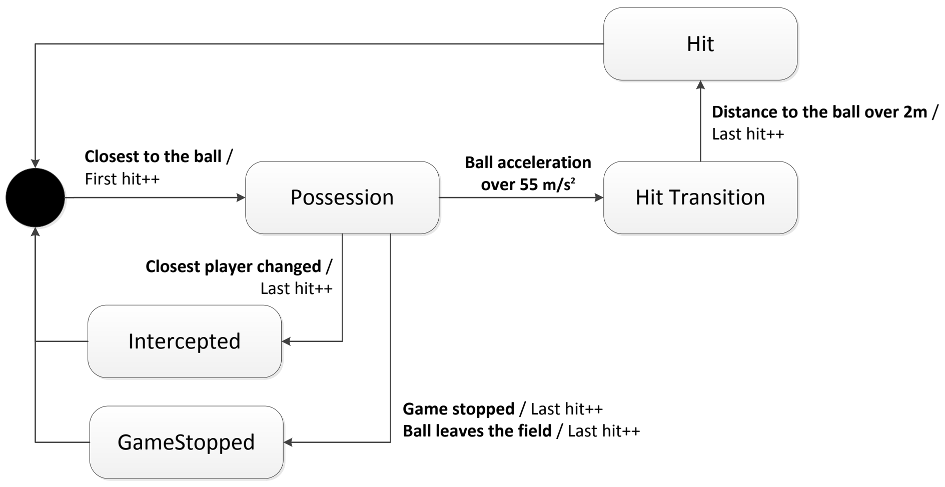
2.4 Implementing Query 2

The statistics needed for query 2 are ball possession of players and teams. To acquire these statistics, we created two processing blocks: *PlayerBallPossessionProcessor* and *TeamBallPossessionProcessor*. In the *PlayerBallPossessionProcessor*, the sensor data of players and balls are subscribed from *SensorDataGenerator* to find a ball possessing player by computing the distance between players and the ball, and the ball's acceleration. The detailed state diagram and algorithm for the player ball possession is described below and shown in Figure 3.

According to the state diagram described in Figure 3, possession for each player is calculated as follows. First, all the players are in the *initial state* which indicates the player does not possess the ball. The player closest to the ball within a 0.3 meter radius at a certain timestamp is calculated using the x and y coordinates of the ball and the players' data. This player has possession of the ball starting at that timestamp and is in *possession state*.

The player gets his last hit while entering the states below as follows:

1. **Hit**: when the distance to the ball is over two meters. When the distance to the ball from the player who possessed the ball is over the limit, it can be assumed that the player passed or shot the ball with the last touch.
2. **Intercepted**: if the player that possessed the ball previously is still in the *possession state* and is not able to reach the *hit state*, and the closest player to the ball is changed, we can interpret this to mean that the ball has been intercepted by another player.



Hit
Distance to the ball over 2m / Last hit++
Closest to the ball / First hit++
Possession
Ball acceleration over 55 m/s²
Hit Transition
Closest player changed / Last hit++
Intercepted
Game stopped / Last hit++
Ball leaves the field / Last hit++
GameStopped

Fig. 3. State Diagram for PlayerBallPossessionProcessor

3. **GameStopped**: the game is *stopped* (including when the ball leaves the field) when the x and y coordinates of the ball are not in the field or when the ball's sensor timestamp is not in the time range of first or second half.

Moreover, if the ball's height exceeds 1 meter, then subsequent players that may be closest are ignored, i.e, if the ball is in flight, no one has possession of the ball till its height is below 1 meter. For a team possession, we have a separate processor called *TeamBallPossession*, which subscribes to the result stream of the player ball possession. We are calculating it by storing the possession time of each player at every timestamp into a double-ended queue. To display the team possession at the current timestamp, the possession for each player on the team is aggregated from the current timestamp, which is located at the end of the appropriate queue with the previous timestamp.A team's possession percentage is calculated accordingly. As the query incorporates the four time windows, the processor publishes four result streams by different data writers.

2.5 Implementing Query 3

Query 3 is perhaps the most straightforward query in its logic. However, the amount of data required to be kept in the system to accurately calculate the results quickly grows and needs to be handled properly. To do this, we have a receiving node, called *AccumulatingHeatMapProcesser* (Accumulator), processing player positions from the *AverageProcessor* at 50Hz. This Accumulator reads in the location data for each player and builds a logical data structure of 6,401 integers: the total number of samples that were recorded for that second, and then values for each of the 64x100 row/column cell value.

Our solution incurs slightly inaccurate approximations on the 8x13 grid of cells because the 100 rows of the 64x100 cell matrix cannot be divided evenly into a 8x13 matrix. The result is that our 8x13 matrix cells are shorter by 1/2 of a 64x100 cell for each of the 8 cells. This was made to simplify the data processing and decrease the data transmission requirements. This can be easily updated if exact precision is required. This solution allows the same algorithm to determine the appropriate result stream for all 4 grid sizes by accumulating the 4 cells from the next more accurate data for each required cell of the result stream.

Sample data per player is then subscribed to by the *HeatMapProcessor* processor block every second. This processor block acts as an accumulator and maintains the data which will be used for producing the 16 result streams for each player. It uses the 64x100 grid to mathematically generate, by adding the appropriate 2x2 cells from the next larger cell matrix to obtain the 32x50, 16x25, and 8x13 grid of cells when that information is needed. This node could be configured in a production system to actually be a single node per player, but we have found that it works well as a single node for the data provided.

The processor block keeps track of this data by building a running tally of all 16 desired output streams: 4 time windows * 4 grid sizes. By doing this, the node has to compute only two functions per stream. First, it has to subtract the older data that no longer fits into the appropriate time window from the running tallies in the 12 streams where the entire game data is not desired. Then, the processor block has to add the newest one second worth of accumulated data to each of the streams' tallies.

There are two ways to implement different time windows utilizing DDS middleware. First, each processor keeps history values in the processor's cache and calculates a proper value using the history values according to the size of the time window. Second, the history values can be kept and managed in the middleware cache by making use of *History* QoS and *Resource Limits* QoS. In query 3, we adopted the latter approach. We approximated the size of middleware cache in our configuration of the QoS such as history depth and maximum number of samples according to arrival rate of input streams. In the case of query 3, the arrival rate is 1 second and requires a minimum cache memory storage of 60 samples for 1 minute, 300 samples for 5 minutes, and 600 samples for 10 minutes. However, history samples for 1 minute and 5 minutes are actually overlapped with history samples for 10 minutes, so we allocate memory for history samples for 20 minutes and other time windows reuse the same memory, which is a tradeoff and an optimization.

2.6 Implementing Query 4

The query 4's goal is to detect events of a player shooting in the direction of the goal. Shots on goal occur when a player hits the ball (i.e. his foot is close to the ball and the ball's acceleration is greater than 55 m/s^2) and the projection of the ball's movement would lead it within one of the two goal areas. Since detection of hit events is already carried out to compute query 2, *ShotsOnGoalProcessor* subscribes to a data-stream published by by *PlayerBallPossessionProcessor*. Similary, the filtered data-stream reporting data about the ball currently used in the game is used instead of raw sensor data.

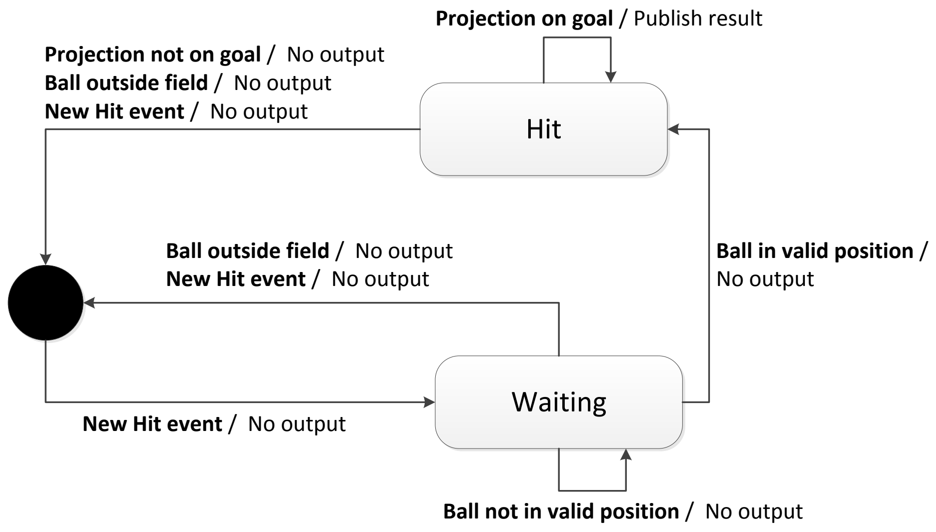

Projection on goal / Publish result
Projection not on goal / No output
Ball outside field / No output
New Hit event / No output
Hit
Ball outside field / No output
New Hit event / No output
Ball in valid position / No output
Waiting
New Hit event / No output
Ball not in valid position / No output

Fig. 4. State Diagram for ShotsOnGoalProcessor

Detection of shots on goal is implemented using a simple finite-state machine represented in Figure 4. The processor uses asynchronous notifications from DDS to wait for input data to be available. The latest hit event and ball position samples received are stored by the processor between successive notifications. Whenever inputs are updated, the state-machine is updated. Once a hit event is notified, the processor must wait for the ball to have moved sufficiently farther from the initial location of the hit so that its motion can be projected more accurately using the equations for projectile motion.

Whenever the processor detects a valid projection on the goal, it will publish a result sample containing the shooting player's id and the ball's position at the time of detection. In this implementation we used simple 2-dimensional equations for these computations, which seemed to yield accurate enough results. Nevertheless, it is trivial to replace the projection calculating algorithm if results are found to be not satisfactory. The results are produced until the projection is found to be missing the goal with updated ball position information, in which case the processor resets its state.

The state-machine is also transitioned back to the initial state every time the ball ends up outside the field boundaries or a new hit event from a different player is received. Depending on the current values of the inputs, multiple state transitions may occur with a single state update. This enables the processor to consume available input more efficiently and avoid unneccessary waiting when the ball is already in a valid position to compute its projection toward the goal. For example, in the case of a new hit event, the processor may execute a maximum of 4 consecutive state transitions within a single state update, if the ball is immediately found within valid range for computing the projection of the new hit event and the processor determines the ball will not hit the goal.

3 Evaluation

In this section, we experimented with two aspects of performance of our system. First, data update rates of a result stream are measured to show efficient resource management within given deadlines by making use of DDS QoS. Second, to show scalability of the system, we measured processing latency of a query stream as the number of clients subscribing to a result stream is increased. We used the *current running analysis* query because results of the query are updated continuously and at high frequency. Our configuration is as follows. One machine is used for sensor data generation and query processing for both experiments. In the update rate experiment, a machine is used for a subscribing client process and, ffor the scalability testing (to provide as many client processes as are needed to generate loads to our system), we used 5 client machines. The following is the hardware specification of the machine: 12-core 2.1 GHz Opteron CPUs, 32GB of memory, and Gigabit Ethernet cards. All physical host machines are connected by a Gigabit Ethernet switch. RTI Connext Professional Edition 5.0 is used for the DDS implementation.

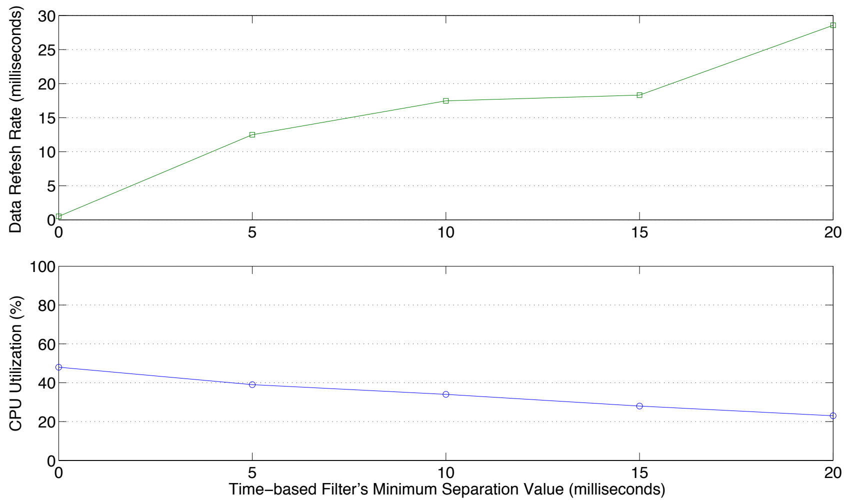


Fig. 5. Data Refresh Rates Per a Player and CPU Utilization by Different Time-based Filter Values

Figure 5 shows data refresh rates per a player and CPU utilization of the serving processor, *CurrentRunningAnalyzer*. Through this experiment, a clever way to manage restrained resources for soft real-time systems by making use of *Time-based Filter* QoS is introduced. According to the requirement of the DESB Grand Challenge, the maximum frequency of this query is 50Hz and therefore the update rate of this query should be 20 milliseconds. As the graph shows, when we set the minimum separation value to 10 milliseconds and 15 milliseconds, averaged latency is around 17 milliseconds and 18 milliseconds which actually satisfy the requirement, respectively. If 20 milliseconds is the deadline, the result with 10 milliseconds of minimum separation is turned out that only 60 out of 100000 samples miss the deadline (0.06% of deadline-missing ratio), but CPU utilization is reduced about 10 to 20% comparing the one without time-based filter.

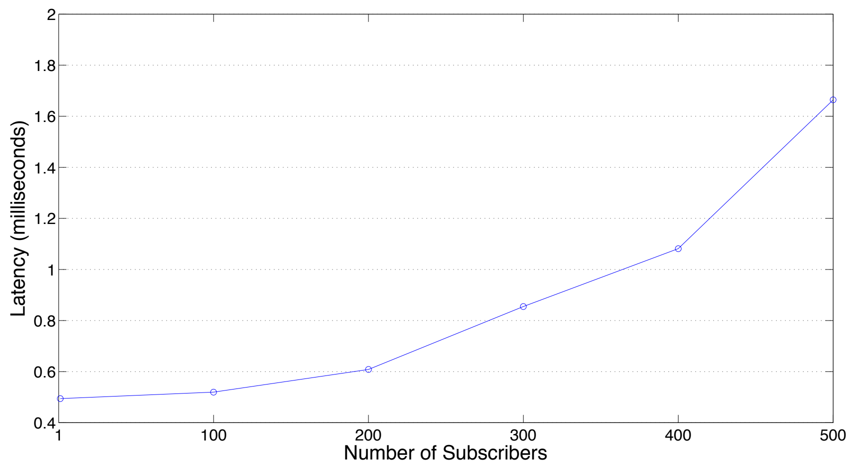


Fig. 6. Processing Latency by Increasing Number of Subscribers

To measure complete latency of the query, a starting timestamp is recorded when a sample is fed into our system. Then, the timestamp is delivered within a message to a client processor and an ending timestamp is logged at the client processor to calculate the difference between two timestamps. However, in the experiment, we deployed the query processors and client processors in separate machines where system time is not synchronized and therefore accurate latency cannot be measure. To resolve the issue, Precision Time Protocol (PTP) [4] was exploited to guarantee fine-grained time synchronization between different physical hosts as it achieves clock accuracy in the sub-microsecond range on a local area network.

Figure 6 shows the processing latency of the current running analysis of query 1. We tested our system using an increasing number of subscriber processes in order to verify its scalability. For better performance, we partitioned processing data by *player_id* and fed each separate data stream into parallel processors, each one responsible for a different set of players. DDS filtering mechanisms facilitate partitioning by leveraging the concept of *instances* identified by a *key* (in this case, *player_id*). Each instance can be processed independently of the other. Data flow to each element of the processing architecture can be controlled without requiring modifications to its code by leveragin XML-based DDS configuration.

The processing latency starts at 500 microseconds, for a single subscriber, and it increases linearly to 1.6 milliseconds, when serving 500 subscribers. Additional scalability could be added to the system by adding multiple DDS domains to the processing architecture. Each domain will actively replicate all (or some of) the processing functionalities and cater a separate set of end clients.

4 Concluding Remarks

This paper described our solution to the DEBS 2013 Grand Challenge Problem. Our solution implements the four queries that require real-time stream processing and analytics using the OMG Data Distribution Service (DDS) middleware and a variety of its QoS policies. By using DDS, we were able to architect our solution in a way where we could focus on developing the algorithms to address each query while rely on the middleware to provide us out-of-the-box solutions needed for distribution, timeliness and reliability, in-network database, and scalability to accommodate a large number of entities.

References

1. Jerzak, Z., Ziekow, H.: The acm debs 2013 grand challenge. `http://www.orgs.ttu.edu/debs2013/index.php?goto=cfchallengedetails` (2013)
2. OMG: The data distribution service specification, v1.2. `http://www.omg.org/spec/DDS/1.2` (2007)
3. OMG: Extensible and dynamic topic types for dds (dds-xtypes). `http://www.omg.org/spec/DDS-XTypes` (2012)
4. Carroll, L.: Ieee 1588 precision time protocol (ptp). `http://www.eecis.udel.edu/~mills/ptp.html` (2012)