Using Binary Trees to Synchronize Events in Heterogeneous Datastreams

Ștefan-Szalai Dragoș* and Mircea-Florin Vaida*
* Faculty of Electronics, Telecommunications and Information Technology/Communications Department, Cluj-Napoca, Romania

Abstract— **In the context of growing ubiquity of sensors, surveillance equipment and other mobile devices, a shift in the data processing paradigm was necessary. New systems are required to be capable of processing data streams of infinite length, having a high throughput, that cannot be stored and processed using classical Database Management Systems (DBMSs). These are called Data Stream Management Systems (DSMSs) within the scientific community. A first step performed by them is time synchronization between events arriving on different timestamped data streams. Within this paper an event synchronization method that makes use of binary trees to achieve its task is introduced and compared with other approaches in order to emphasize its strengths. Furthermore the integration with DSCPE (our Data Stream Continuous Processing Engine) is proposed.**

I. INTRODUCTION

The latest advances in science coupled with the ability to build very small devices have led us into a world where monitoring equipment is ubiquitous. It is not uncommon to see such equipment in the list of assets belonging to small companies that could not have afforded such devices a couple of years ago.

On the market several types of sensors and monitoring equipment can be found, the most common being the ones needed for building surveillance, for detecting various changes related to the surrounding climate, monitoring patients in hospitals and others. All these are appliances that generate data under the form of data streams. Financial data is exposed by stock exchanges over low latency and high throughput network links as data streams, and network traffic is subject to collection for further analysis in some nodes. According to *mobithinking.com*, in the entire world there are 5981 million mobile cellular subscriptions, a huge number when taking into account the entire world population, all of which are sources of data.

To our understanding, data streams represent infinite sequences of events that are accompanied by a timestamp and are ordered in time.

Considering the amount of data coming from these systems it does not take long to realize that the classical *store and query* approach cannot be applied. DBMSs (DataBase Management Systems) cannot work on infinite streams because in order to query they need to have access to the entire set of data. A list of problems researchers are facing can be found in L. Golab and M.T. Özsu, [1].

In response to the challenge imposed by infinite data streams, researchers developed *continuous query systems* [2] [3] [4], defined *dataspaces* [5] [6], and gave birth to a new class of systems, called *DataStream Management Systems* (DSMSs) [7] [8] [9], that are capable of querying received information on the fly and of providing the user with an approximate result of the query execution.

DSMSs are not the only ones that work on infinite streams. Datastream mining tools are under development that are capable of extracting information related to analyzed datastreams without requiring access to the entire history of data. Some of the directions aimed when building these tools are:

- Data stream mining utilities for classification, like the one found in P. Domingos and G. Hulten, [10].
- Dependency detection applications like the ones in T. Oates et al. [11], H. Kargupta et al. [12].
- Different tools for anomaly and fraud detection like the ones in A. Metwally et al. [13] and A. Pawling et al. [14].

We call the entire set of applications that work on datastreams *continuous processing applications*.

Time synchronization is an important issue in these systems as most of time they base their results partly on the temporal information conveyed through the data streams, and the information travelling between the monitoring equipment and the processing system is sent over links that can be subject to different amounts of delay. Even though it is important, the time synchronization subject is either avoided or not thoroughly explained in the work analyzed. When taking a look at NiagaraCQ, presented in [2], we observe that time synchronization is not addressed. TelegraphCQ, [4], allows loosely synchronized sources of data, meaning that a synchronization is not performed prior to applying query operators. Aurora and Medusa, [7], perform reordering within the query operators and a deadlock prevention mechanism is said to be implemented but is not presented in the paper. The in-query operator implementation is of no use in our dependency detection project. STREAM, [8], performs synchronization when a join operator is applied to data streams. The process is not presented in detail. Borealis, [9], is based on Aurora and Medusa but uses a different approach for synchronization. It uses a revision based system within which any messages that exceed imposed delay bounds are dropped. The description of this system is not detailed.

Other papers that have been studied in the search for a time synchronization solutions are Sun et al., [15], which describes a burst detection algorithm on data streams, Das et al., [16], where a way of predicting the future actions of the inhabitants in a smart home can be found, Dechouniotis et al., [17], where a method to detect dependencies between network components using a fuzzy algorithm is presented. None of these presents a usable way of synchronizing events within our solution.

In this paper we propose two methods for synchronizing events that use binary trees as data structures for performing the task. The algorithms employed are explained, the results of the two methods are compared and the integration with our continuous processing engine is described.

The remainder of this paper is organized as follows: section II presents the definitions of the main concepts that are going to be used, section III depicts the problem, section IV describes the proposed methods and tests performed on them and section V presents future directions and concludes this paper.

II. DEFINITIONS

Definition. A *data event* $E(t, a_1, \dots, a_k)$ is a tuple containing a timestamp t and a list of attributes $a_1, \dots, a_k$ that describe the state of the event source.

Observation. Attributes are not required for the work presented in this paper, so they will be omitted from now on when mentioning events.

Definition. A *data stream* DS is a sequence of timestamped data events produced by a data source ds.

It is assumed that a data source cannot be in more than one state at a time (unique timestamp) and timestamps are considered as a monotone sequence of positive integers (*discrete-time*).

$$DS = \{E_i(t_i) | t_i \in \mathbb{N}, t_i < t_{i+1}\} \tag{1}$$

Within 1, t_i represents the timestamp of the event E_i.

Let $DS_1, DS_2, \dots, DS_n$ be the data streams being analysed in a continuous event processing system, where n represents the number of streams.

Observation. For events that correspond to a data stream DS_k the notation $E_i^{DS_k}$ is used in order to avoid confusion between events from different data streams.

For the remainder of the paper it is assumed that the source clocks being used to timestamp the events are synchronized and that the only source of time variation is the transportation of events between the data source and the processing system. Moreover it is considered that events produced by the same data source will arrive in order at the processing system.

An event $E_i^{DS_k}$ that arrives at the processing system receives an *arrival timestamp* ta_i.

Definition. The tuple formed by associating an arrival timestamp ta_i to an event $E_i^{DS_k}$ is called an *arrived event* and is denoted by $AE_i^{DS_k}$.

$$AE_i^{DS_k} = (ta_i, E_i^{DS_k}) = (ta_i, t_i) \tag{2}$$

Let SAE be the *set of all arrived events* within the processing system:

$$SAE = \bigcup_{k=(1,n)} AE_i^{DS_k} \tag{3}$$

Definition. An arrived event $\mathrm{AE}_j^{(\mathrm{DS_k})}(\mathrm{ta_j}, \mathrm{t_j})$ that has the properties described in 4.1 and 4.2 is called a *delayed event.*

$$ta_j \geq ta_p, \forall\, AE_p \in SAE \tag{4.1}$$

$$\exists\, AE_q \in SAE, s.t.\ t_j < t_q \tag{4.2}$$

Using the definitions and observations specified above the problem can be defined and our solution approach explained.

III. PROBLEM STATEMENT

A. Time synchronization

Given a set of datastreams to be processed by DSCPE (Data Stream Continuous Processing Engine), see [18], on which events can be subject to delays on their path from the source to the system, we are interested in creating a solution that would synchronize the arrived events in order to ensure that processing is done in proper order.

Figure 1 illustrates DSCPE and the place where time synchronization is needed. It is important to note that the information received on different streams is represented as events that have the same structure. More explanations related to DSCPE will be given later within this paper.

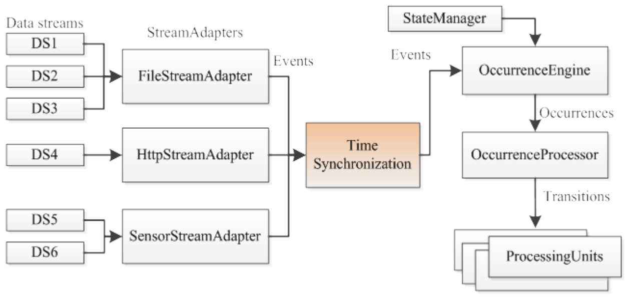

Figure 1. Where time sync is needed in DSCPE

Figure 2 illustrates an example of delayed event arrivals. In order for a proper synchronization to take place, assuming that the synchronization begins at time 0, the processing of events can only begin after event number 9 arrives at the processing system.

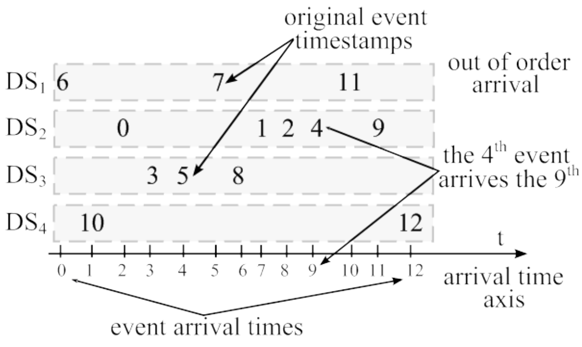

Figure 2. Delayed event arrivals

B. System evolutivity

The number of processed datastreams could change significantly with time in a continuous processing system. In order to cope with this the proposed synchronization method should be able to support changes in the number of synchronized data streams while running.

C. *Datastream Failures*

Given the fact that a stream may fail or stop generating events we are interested in adjusting the time synchronization solution so that such streams would not block the processing system.

IV. Solution Approach

A. *Why binary trees?*

Binary trees have been chosen over other methods because they provided a simple and straight forward manipulation with low time complexity. The insertion, removal and search operations have a worst case complexity of O(n), while the average is O(log(n)). This can mean a synchronization that's sometimes close but always better than the studied alternatives.

Table 1 presents a small comparison based on the number of comparison and update operations performed by possible alternative methods when synchronizing the events in figure 2.

TABLE I

COMPARISON BETWEEN POSSIBLE SYNCHRONIZATION METHODS

	Indexed list	Linked list	Binary tree
Comparisons	25	37	30
Updates	39	37	17

B. *Method 1*

The method described in this section is based on the work presented by us in the paper "SYNCY: A Software Engine for Data Stream Event Synchronization", see [19].

In order to address the ***first requirement*** presented in section III, subsection A, a binary tree structure was chosen having the following characteristics:

- The arrived events are stored as nodes that can have a maximum of 2 descendants (one left descendant and one right descendant).
- The insertion position of each node is determined by the original timestamp of the arrived event. If it's smaller than the one in the current node we'll continue the insertion on the left branch, otherwise on the right.
- Besides the 2 descendants, each node contains a link to its parent in order to allow non-recursive implementations for insertion and removal operations within the binary tree.
- Each node contains a bitwise table where 1 bit is allocated for each datastream that the system processes at the time of insertion. When an event newer than the one represented by the node arrives on a datastream the bit corresponding to this datastream will be set to 1. When all bits are set to 1 the node can be processed. We call this bitwise table the *readiness table.*

Within figure 3 we can observe how the events in figure 2 are added to the tree and how readiness tables should be updated.

The ***second requirement***, presented in section III, subsection B, is addressed through proper usage of the readiness tables.

Algorithm 1. AddNodeToTree(*currentNode*, *nodeToAdd*)

```
if nodeToAdd.Event.t < currentNode.Event.t then
    if currentNode.Left != NULL then
        AddNodeToTree (currentNode.Left, nodeToAdd)
    else
        currentNode.Left = nodeToAdd
        nodeToAdd.Parent = currentNode
    end if
else
    currentNode.
        SetFlagForDataStream(nodeToAdd.Event.DS)
    if currentNode.Right != NULL then
        AddNodeToTree(currentNode.Right, nodeToAdd)
    else
        currentNode.Right = nodeToAdd
        nodeToAdd.Parent = currentNode
    end if
end if
```

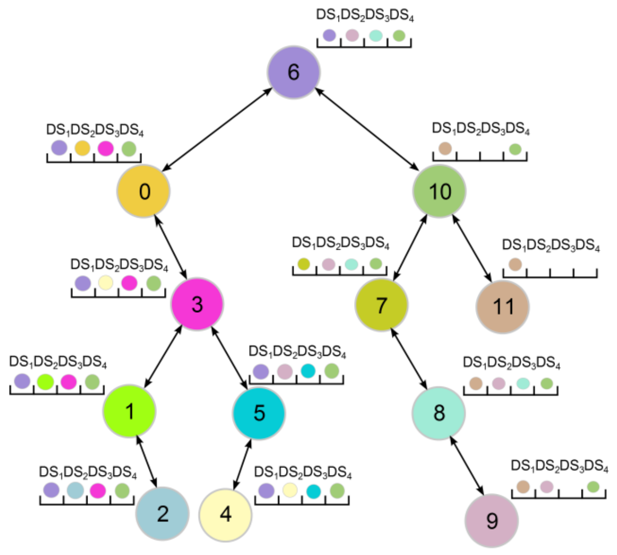

Figure 3. Example of binary tree

Each time a new stream is added to the system the readiness table increases in size for the newly created nodes. When comparing readiness tables of different sizes only the common part is taken into account during calculations.

When a stream is removed from the system its position will be masked in the readiness tables in order not to affect processing.

Addressing the ***third requirement*** (section III, subsection C) implies the definition of two important parameters, the maximum allowed processing delay for an event, or *MD*, and the maximum allowed failures per stream, or *MS*.

Each event to be processed receives an arrival time stamp. If the current system time minus the arrival timestamp of an event that is still in the tree is greater than *MD*, the event is processed at the next event processing step and the failure marker is incremented for the stream that produced that event. When the failure marker for a stream reaches *MS*, the stream is disabled until an event

Algorithm 2. *ProcessTree*(*rootNode*)

```
if rootNode != NULL and
        (IsReadyToProcess(rootNode) or
        IsSyncTimedOut(rootNode))
then
    oldRootNode = rootNode
    rootNode = GetBestNewRootMatch(rootNode)
    SendToProcessingRecursive(oldRootNode)
end if
```

Algorithm 3. *SendToProcessingRecursive*(*currentNode*)

```
if currentNode != NULL
then
    SendToProcessingRecursive(currentNode.Left)
    Enqueue(currentNode)
    SendToProcessingRecursive(currentNode.Right)
end if
```

with an original timestamp newer than the latest one being processed arrives on it.

Events that have an older timestamp than the latest event being sent to processing should be dropped.

The following operations are performed for node addition and synchronization:

Node addition – Algorithm 1 presents the recursive steps performed while adding a node to the tree. The search for the insertion point starts from the root node and continues with its descendants until a descendant in the chosen direction no longer exists and the insertion can take place. While searching for the insertion point we update the readiness tables of the inspected nodes if the process continues on their right branch.

Tree processing – Takes place after the node addition and is presented in algorithm 2. The first step is to check the root node if it has all the bits in the readiness table set to 1 or if the node synchronization has timed out. This means that the node is ready to process. The next step is to find the closest right descendant of the root node that cannot be processed and to set it as the root node. Once this step is done we can proceed with sending the events tot the event processor. The events will be extracted from the partial tree in an in order fashion until none remains.

Algorithm 3 presents the recursive variant of the algorithm employed for node extraction and processing.

The details related to how the maximum allowed processing delay and maximum stream failure are handled have been partially omitted from the code snippets provided. The mechanism behind them is simple and it does not require additional explanation.

C. Method 2

The second method, presented in this section, uses the same tree structure as the first and ads some important changes to the way nodes are added and extracted from the tree.

In an attempt to reduce the *system delay* of the first method, defined as the time required to extract a node that has reached the state ready to process, we joined the tree processing algorithm and the node addition algorithm. The resulting method exploits some of the previously exposed properties of the binary tree, namely:

Algorithm 4. AddNodeToTreeAndProcess (*currentNode, nodeToAdd, canCheck, mask*)

```
if (currentNode == nodeToAdd) then
    if canCheck and IsReadyToProcess(currentNode,
            mask) then
        SendToProcessingRecursive(currentNode)
    return

if nodeToAdd.Event.t < currentNode.Event.t then
    if canCheck then
        UpdateMask (mask, currentNode)
    if currentNode.Left != NULL then
        AddNodeToTreeAndProcess (currentNode.Left,
                                nodeToAdd, canCheck, mask)
    else
        currentNode.Left = nodeToAdd
        nodeToAdd.Parent = currentNode
        AddNodeToTreeAndProcess (currentNode.Left,
                                nodeToAdd, canCheck, mask)
else
    currentNode.
        SetFlagForDataStream(nodeToAdd.Event.DS)
    if canCheck then
        if (IsReadyToProcess(rootNode, mask) or
                IsSyncTimedOut(rootNode)) then
            SendToProcessingRecursive(currentNode)
        else
            canCheck = false
    if currentNode.Right != NULL then
        AddNodeToTreeAndProcess (currentNode.Right,
                                nodeToAdd, canCheck, mask)
    else
        currentNode.Right = nodeToAdd
        nodeToAdd.Parent = currentNode
        AddNodeToTreeAndProcess (currentNode.Right,
                                nodeToAdd, canCheck, mask)
```

1. The left descendants of one node inherit the values from the readiness table of this node.
2. If the current node is not ready, neither are its right descendants.

Within figure 3 we can observe these properties, together with how nodes are added to the tree and how readiness tables are built.

The first property represents a burden as it requires the creation and maintenance of a readiness mask which must be updated every time the left descendant of one node is chosen for inspection.

The second property helps limit the impact of the first by allowing us to skip all extra computations after choosing the right descendant for inspection if the current node is not ready for processing.

Algorithm 4 describes the recursive version of the merged add and process operations. The *end if* instructions were omitted as their positions can be implied by taking into account the indentation.

D. *Integration with DSCPE*

This chapter describes how the synchronization methods should be packaged in order to operate within DSCPE.

In DSCPE the *stream adapters* produce *event* objects within which the arrival timestamp and the original timestamp are stored for represented events. Event objects are fed to an *occurrence engine* which is used to extract the temporal properties from a chain of events and to package the details under the form of *occurrence* of *states.*

When packaging the synchronization solution we must take *events* as input and output *events.* This can easily be done by keeping the correspondent events within the tree nodes and manipulating them as necessary.

But if we want to handle bursts we need to implement an output queue or use an existing one (the STL queue for example). The events stored in an output queue must be dequeued by an autonomous routine that takes them from the queue and sends them to processing.

This is why it is important to mention that a time synchronization module would be comprised of:

1. A time synchronization part
2. An output queue
3. An event feeder that takes events from the queue and sends them to the *occurrence engine.*

A block schema of DSCPE with time synchronization is presented in figure 1.

E. *Testing time sync methods*

In order to test and compare our proposed time synchronization methods we used the same generator as the one used to produce the results in [19].

The following adjustments can be made within the event generator:

- the number of data streams
- the delay and throughput can be changed for the desired streams
- the number of events to generate at each run

The first test performed demonstrates the *impact of the number of data streams on the processing time.*

The number of data streams was varied from 1 to 1500 with all streams having the same throughput and 0 delay. For each case 10000 events where generated and the number of CPU clocks required to complete it was measured together with the number of visited nodes. The result can be seen in figures 4 and 5.

In figure 4 we can see that both method 1 and method 2 require about the same amount of CPU clocks to perform synchronization on streams that are implicitly synchronized. The second method required slightly more than the first, but this is not yet obvious.

The number of visited nodes is smaller for the second method than the first, as we can see in figure 5, but if we look at the values corresponding to a high number of data streams we can conclude that the advantage is negligible. The axes were represented using a logarithmic scale in order to emphasize the difference in the number of visited nodes for small numbers of data streams. On the same figure we can deduce that the increase in number of visited nodes is linear and in a direct relation with the increase in the number of processed datastreams.

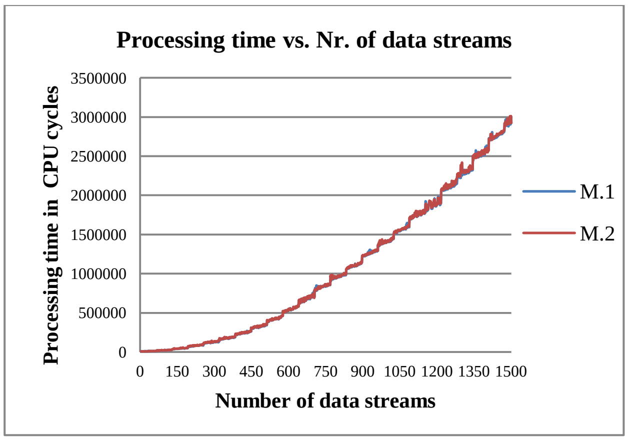

Figure 4. Processing time vs. Number of data streams

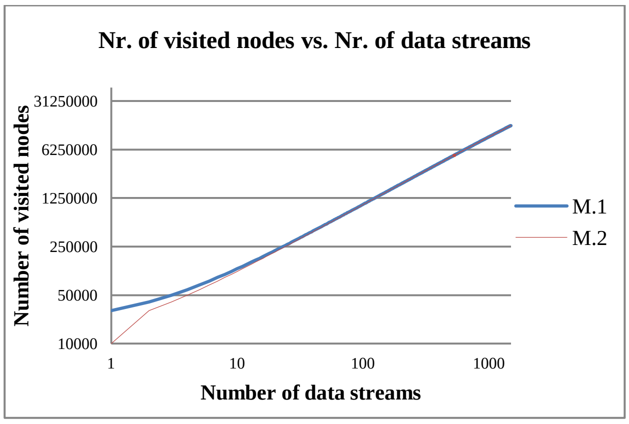

Figure 5. Nr. of visited nodes vs. Nr. of datastreams

Due to the way readiness tables are implemented we can observe a high increase in number of CPU cycles required at each multiple of 64 data streams. The readiness tables are implemented in a bitwise manner having as storage unit the default platform word, which is 64 bits for 64 bit platforms.

The second test performed tries to emphasize the link between the delays occurring on multiple datastreams and the amount of processing required for fulfilling the tasks.

The number of datastreams was varied from 1 to 256 and delays were applied to the first two streams.

The results presented in [19] show that for the first method the synchronization time is only affected by the stream having the highest delay. These results are also valid for the second method since we are dealing with a technique similar to the first one. However, we are interested in comparing the two methods in order to see how they perform in the same conditions.

Figure 6 shows the number of CPU cycles required by each method for different sets of parameters. It is easy to observe that the second method requires more cycles to perform the same task, with a lower latency though, and that the increase in required computation power is not linear with the growth of the number of datastreams to be processed.

Due to the fact that the second method requires more processing power its usage is recommended only in systems where having a very low delay is critical. Otherwise the first presented method will perform well.

Using a balanced tree could improve the second method by lowering the time required to add a node to the tree and thus the number of required calculations. Red-Black trees are a good candidate and further tests will be made in this direction. On the contrary, the first method cannot be improved using a balanced tree as it will break the node extraction mechanism.

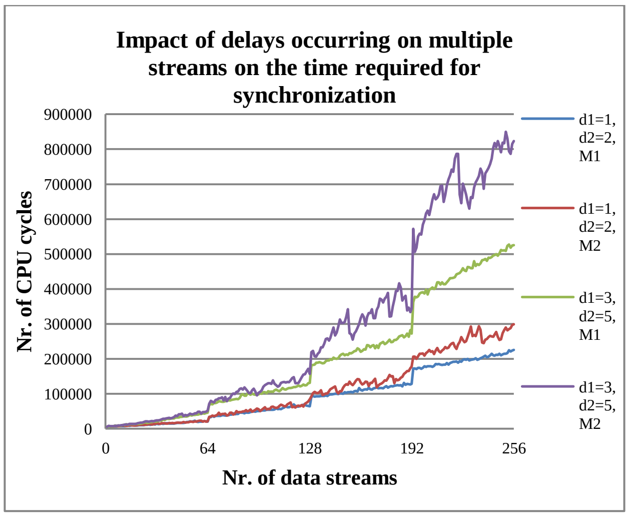

Figure 6 Impact of delays on the nr. of CPU cycles required

V. Conclusions and Future Work

This paper is related to our dependency detection in large surveillance networks project, currently under research. As you have seen within the paper, the current methods are already implemented and used by DSCPE to perform dependency detection.

Improvements to the synchronization methods are planned and the main directions are:

1. *Improvement of memory handling* – synchronizing events implies a fast creation and disposal of objects. These types of operations have an important impact on system performance and thus they are subject to optimization where possible. *Pooling* and *arena allocation* are two concepts that can help improving performance and they will be analyzed and possibly used in the near future.
2. *Multi-threading* – the current implementation is not multi-threading friendly. As systems that work on data streams require multi-threaded operation this is one of our future research and development direction related to data stream synchronization.

We consider that the two methods presented in this paper are a good fit for appliances where the data sources are reliable and synchronized with a common source. Depending on latency and CPU requirements one can choose to use the first method if a small added latency is not an issue or the second if latency is critical and enough processing power can be provided to perform the task.

The integration with DSCPE will motivate us to keep the time synchronization methods up to date and to improve them as previously discussed.

Acknowledgment

This paper was supported by the project “Doctoral studies in engineering sciences for developing the knowledge based society - SIDOC”, contract no. POSDRU/88/1.5/S/60078, project co-funded from the European Social Fund through Sectorial Operational Program Human Resources 2007-2013.

References

[1] L. Golab and M. T. Özsu, "Issues in data stream management," *SIGMOD Rec.*, vol. 32, no. 2, pp. 5-14, 2003.

[2] J. Chen, D. J. Dewitt, F. Tian and Y. Wang, "NiagaraCQ: A scalable continuous query system for Internet databases," *SIGMOD*, pp. 379-390, 2000.

[3] R. Avnur and J. M. Hellerstein, "Eddies: continuously adaptive query processing," *ACM SIGMOD Record*, vol. 29, no. 2, pp. 261 - 272, 2000.

[4] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin and J. M. Hellerstein, "TelegraphCQ: continuous dataflow processing," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, New York, 2003.

[5] M. Franklin, A. Halevy and D. Maier, "From databases to dataspaces: a new abstraction for information management," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 27 - 33, 2005.

[6] M. Franklin, A. Halevy and D. Maier, "A first tutorial on dataspaces," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1516-1517, 2008.

[7] S. Z. Sbz, S. Zdonik, M. Stonebraker, M. Cherniak, U. C. Etintemel, M. Balazinska and H. Balakrishnan, "The aurora and medusa projects," *IEEE Data Engineering Bulletin*, vol. 26, 2003.

[8] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, K. Ito, R. Motwani, U. Srivastava and J. Widom, "STREAM: The stanford data stream management system," *Springer*, 2004.

[9] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniak, J. Hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, R. Ryvkina, N. Tatbul, Y. Xing and S. Zdonik, "The design of the borealis stream processing engine," *CIDR*, Vols. 277-289, 2005.

[10] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proceedings of the sixth ACM SIGKDD international conference*, New York, 2000.

[11] T. Oates, M. D. Schmill, D. E. Gregory and P. R. Cohen, "Detecting complex dependencies in categorical data," 1994.

[12] H. Kargupta, K. Sivakumar and S. Ghosh, "A random matrix-based approach for dependency detection from data streams," in *DMKD*, 2002.

[13] A. Metwally, D. Agrawal and A. E. Abbadi, "Using association rules for fraud detection in web advertising networks," in *Proceedings of the 31st international conference on very large data bases*, 2005.

[14] A. Pawling, P. Yan, J. Candia, T. Schoenharl and G. Madey, "Anomaly detection in streaming sensor data," in *IJCAI*, 2008.

[15] A. Sun, D. D. Zeng and H. Chen, "Burst detection from multiple data streams: a network-based approach," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 40, no. 3, pp. 258-267, 2010.

[16] S. K. Das and D. J. Cook, "Designing and modeling smart environments (invited paper)," in *Proceedings of the 2006 international symposium on World of Wireless, Mobile and Multimedia Networks*, Washington D.C., 2006.

[17] D. Dechouniotis, X. Dimitropoulos, A. Kind and S. Denazis, "Dependency detection unsing a fuzzy engine," in *Proceedings of the Distributed systems: operations and management 18th international conference on managing virtualization of networks and services*, Berlin, 2007.

[18] S. Ş. Dragoş and M. F. Vaida, "DSCPE - A Data Stream Continuous Processing Engine," in *SICOM*, Cluj-Napoca, 2012.

[19] S. Ş. Dragoş, M. F. Vaida, L. A. Şuta, M. C. Ureche and A. Voina, "SYNCY: A Software Engine for Data Stream Event Synchronization," in *Proceedings of the 16th International Conference on System Theory, Control and Computing (ICSTCC)*, Sinaia, 2012.