*I.J. Information Engineering and Electronic Business*, 2016, 1, 21-28
Published Online January 2016 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijieeb.2016.01.03

# An SDDS-Based Architecture for a Real-Time Data Store

**Maciej Lasota**
Department of Computer Science, Kielce University of Technology, Kielce, Poland
E-mail: m.lasota@tu.kielce.pl

**Stanisław Deniziak**
Department of Computer Science, Kielce University of Technology, Kielce, Poland
E-mail: s.deniziak@tu.kielce.pl

**Arkadiusz Chrobot**
Department of Computer Science, Kielce University of Technology, Kielce, Poland
E-mail: a.chrobot@tu.kielce.pl

***Abstract***—Recent prognoses about the future of Internet of Things and Internet Services show growing demand for an efficient processing of huge amounts of data within strict time limits. First of all, a real-time data store is necessary to fulfill that requirement. One of the most promising architecture that is able to efficiently store large volumes of data in distributed environment is SDDS (Scalable Distributed Data Structure). In this paper we present SDDS $LH^*_{RT}$, an architecture that is suitable for real-time applications. We assume that deadlines, defining the data validity, are associated with real-time requests. In the data store a real-time scheduling strategy is applied to determine the order of processing the requests. Experimental results shows that our approach significantly improves the storage Quality-of-service in a real-time environment.

***Index Terms***—Real-time, distributed data store, scalable distributed data structures.

## I. Introduction

The efficient storing and processing of data becomes one of the most crucial problems in modern IT systems. Emerging technologies brought up new challenges for data stores. Cloud computing [1, 2] offers a new business model called Storage as a Service and causes rapid progress in distributed data stores. Internet of Things (IoT) [3] causes the explosion of machine-generated data with large diversity, from large image files to simple sensor data. Big Data analytics requires fast access to large amount of data. Thus the organization of the data store should be suitable to the method of data processing.

The majority of contemporary data repositories is built with the purpose to store data for off-line processing. The Relational Database Management Systems (RDBMs) are in a common use, however a recent grow of interest in Big Data processing environments caused a significant development of other data store models like NoSQL or NewSQL.

The increasing number of Internet of Things (IoT) devices and Internet services introduces a new category of data sets, the Fast Data [4, 5]. They are characterized not only by a high volume but also by a high velocity.

In Fast Data every data set and every request concerning the data have attributed validity that starts to decrease after a given deadline. Proper storing and processing of such data may thus require a real-time approach. The consequences of missing a deadline depend on the type of a real-time policy [6 - 8]:

- In a hard real-time policy a negative value of validity is assigned to tardy requests and data,
- In a firm real-time policy tardy requests and data have no validity,
- In a soft real-time policy the validity of tardy requests and data is still positive, but it diminishes over time.

The problem of Fast Data storing and processing was addressed in several data store implementations, however most of them do not enforce any time restrictions on that operations. They usually define the real-time as "as fast as possible". In this paper we introduce SDDS $LH^*_{RT}$, a Scalable Distributed Data Structure (SDDS)-based architecture for data store that adheres to the firm real-time policy. The data validity is specified by the deadline associated with the request. The SDDS server schedules requests using real-time scheduling method. Experimental results showed that in our approach significantly more requests may be served in the required time, than in existing data stores, where requests are processed in the FIFO order. In this way a significantly better storage Quality of Service (QoS) may be achieved.

The paper is organized as follows. Section 2 contains a short summary of related work. Section 3 describes the concept of SDDS and in Section 4 a motivation for the research is given. An implementation of real-time policy for SDDS is described in Section 5. Results of testing the

 

prototype implementation of SDDS LH*$_{RT}$ are presented in Section 6. The paper is concluded in Section 7.

## II. Related Work

An RDBM that supports a real-time request processing is called a Real-Time Database System (RTDBS). Many such systems were built for both research and commercial purposes [6 - 8]. Some of them offer only partial real-time functionality while others provide a full support [8].

The real-time data stores are a less explored subject. The majority of such systems is built with the purpose of storing as quickly as possible huge amounts of real-time data. In that context real-time means "happening recently".

VoltDB [5] is an in-memory scalable relational database designed for handling Fast Data. Some large companies like Ericsson and HP take advantage of VoltDB. Similarly to other RDBMs, VoltDB stores data in tables. However, these tables are partitioned column-wise and distributed together with stored procedures over nodes of a cluster. Every stored procedure is executed in a single thread, hence it does not require locking or synchronization. That allows VoltDB to process many queries in parallel, which greatly contributes to its efficiency. To further improve its performance VoltDB replicates some of the most frequently used tables. Despite all of these features, the VoltDB does not guarantee that requests from clients will be processed within predictable time periods.

The Mahanaxar [9] is an ongoing research project, which goal is to build a data storage for intercepting, evaluating and storing real-time data for latter processing. Unfortunately many details about this effort are not yet known.

Druid [10] is an in-memory real-time data store used by such companies like Netflix or eBay. Its architecture consists of four main elements:

- Real-Time Nodes are responsible for ingesting data about events that happened in a given period of time; these data are immediately available for clients,
- Historical Nodes store previously acquired data about past events,
- Broker Nodes direct requests to Historical or Real-Time Nodes,
- Coordinator Nodes distribute and manage Historical Nodes.

Aside from these elements Druid also utilizes external components like databases and file systems. It is built to be high available and fault-tolerant. However, its main drawback, as far as in real-time applications are considered, is significant variability in request processing time (latency).

An interesting real-time distributed storage was developed for gathering and processing data from Phasor Measurement Units used in Wide-Area Measurement System (WAMS) for power grids [11]. The data store is based on Chord protocol which basic version is well-suited for the needs of WAMS. The real-time requirements are met by recognizing a pattern in which periodic and aperiodic tasks appear and by applying a cyclic executive which serves those tasks within given time intervals called time frames. Periodic tasks are given precedence before aperiodic ones. Failing to serve a task of any kind in a given time frame results in postponing its execution to the next time frame. That means that the system adheres to soft real-time requirements.

## III. Scalable Distributed Data Structures (SDDSs)

Multicomputer systems are often used in applications that need huge data storage with a short access time. Local hard disks are not sufficient in such situations. Scalable Distributed Data Structures (SDDSs) are a family of data structures designed for efficient data management in a multicomputer [12-14]. The basic data unit in SDDS may be either a record or an object with a unique key. Those data are organized in larger structures called servers/buckets and stored usually in RAM of multicomputer nodes that run an SDDS server software and connected each other with the Fast Local Area Network (Gigabit or Infiniband). Data from a bucket are saved on a hard disk only when necessary e.g. when the server is shutting down. All of the buckets form an SDDS file (Figure 1).

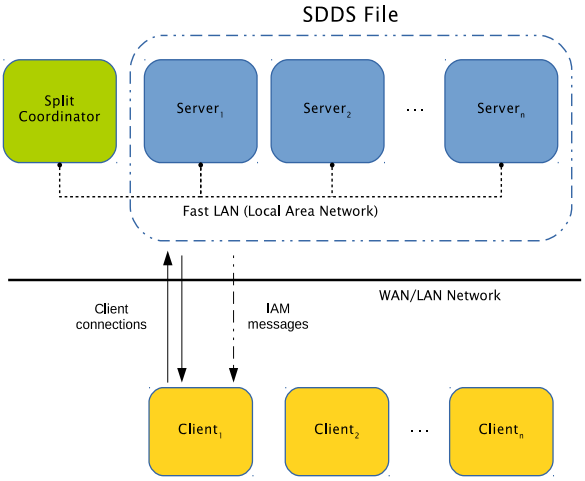


Fig.1. SDDS Architecture

Initially, SDDS file consists of only one bucket which level is equal to zero. When the bucket reaches its capacity limit a collision occurs. SDDS adjusts its capacity to current needs by splitting buckets. When a bucket is overloaded it sends a message to the Split Coordinator (SC) [12, 14, 15]. The SC takes a decision which bucket should split. During a split operation a new bucket is created with the level higher by one than the splitting bucket. Next, about half of the data stored in the splitting bucket is moved to the new bucket. When the transfer is completed the splitting bucket increments its level by one and acknowledges the SC. After the split, both buckets have the same level (Figure 2).

 

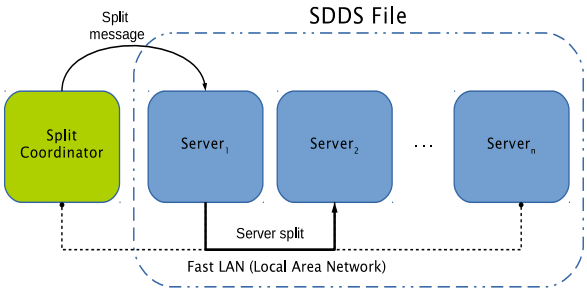


Fig.2. SDDS Server Split

The data in the SDDS file are accessed by an SDDS client software executing on multicomputer nodes, other than servers. The client does not have the whole information about the file. Instead it has its own file image, which may differ from the actual image of the file. The client uses a special function to address data item (a record or an object) in buckets. SDDS may be classified according to addressing functions they use, for example:

- LH* - the client uses linear hashing [12],
- RP* - the client uses range partitioning [13].

Every client has its private image of SDDS file. After the split this image becomes outdated and the client may send request to an incorrect bucket. In such situations the message is forwarded to the correct bucket and the client receives Image Adjustment Message (IAM). The IAM contains new parameters for client's addressing function, that updates its SDDS image.

There is no single point of failure in SDDS, except for the SC, because no central directory is used for addressing. However, there are SDDS architectures (like RP) that do not require the SC. All SDDS implementations follow the same designs rules [12 - 15]:

1) For performance reasons, no central directory is used by the clients in the process of data addressing.
2) The SDDS file adjusts its size to the clients needs by splitting buckets, i.e. moving about half of the content of buckets that reached their capacity to the newly created buckets.
3) Due to the split operations the client's image may become outdated, but it is corrected only when the client makes an addressing error.
4) None of the basic operations on data requires immediate, atomic updating of client's image.
5) If the client incorrectly addresses data, then an Image Adjusting Message (IAM) that allows it to update its image is sent, and the data are forwarded to the correct server.

The SDDSs constitute an example of NoSQL data store design. The data items are stored in-memory in a key-value form. However, the original concept of SDDS is not entirely suitable for Fast Data processing. It lacks the real-time capabilities.

## IV. Motivation

Aside of Fast Data processing there are many applications that could benefit from using a real-time data store [16, 17]. For example, cloud computing environments offer services that typically follow the soft real-time policy by monitoring Quality of Service (QoS) and dynamically allocating resources to applications that are critical for clients [18 - 20]. Using a real-time data storage could simplify this task. Also the automated trading [21], financial [22] and surveillance [23] systems require an access to relevant data in a limited time [6].

The architecture of SDDS is relatively simple and easily to modify, when compared to such data stores like Druid [10]. In our research we have addressed the issue of enabling firm real-time requests of data fetching from SDDS LH*. Such operations are more critical for real-time applications than data write requests.

We have assumed that the data store is used by many clients with different time requirements. This is a common case in many systems. For example, in a large scale medical information system access to the data about intensive care patients should be of higher priority than the access to information about patients who undergo a long term diagnostic procedure. Here, the priority is based on combination of a deadline and service time of a request. Without a real-time scheduling the requests with a longer deadline and time of service (ToS) would postpone the more time constrained ones. As a result they would fail to complete before their deadlines. Figure 3 illustrates such a situation. When requests will be serviced according to the first-in first-out rule (Fig.3 (a)) then only the deadline for request Rq1 will satisfied. If all requests will be reordered according to their deadlines then all time requirements will be met (Fig.3 (b)).

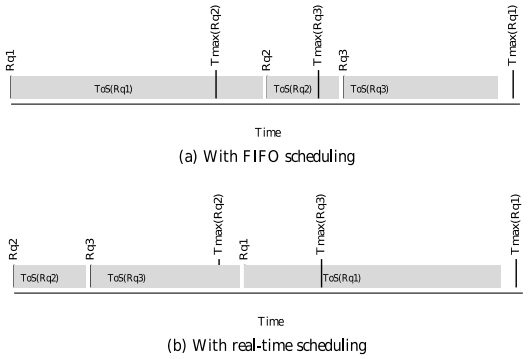


Fig.3. Handling of Requests

We have also assumed that the time of transferring request through network may be properly managed or, at least to some extend, predicted [24].

In our approach we have chosen SDDS LH* [11]. This type of SDDS uses distributed linear hashing for addressing buckets. This algorithm requires an additional element in SDDS architecture called Split Coordinator for overseeing the order in which buckets split. The split operation is performed only after a write requests. Hence providing firm real-time writes for such an SDDS would be more challenging than in the case of any other version

 

of Scalable Distributed Data Structure. On the other hand, it would also give a more general solution to the problem. We also do not discuss in the paper the read requests that result in IAM messages. Those may be prevented by allowing the clients to use filled-up read-only SDDS LH* or by permitting only those writes that modify data instead of adding them. However, we want to address both of the aforementioned issues in our future works.

## V. Implementation of Real-Time Policy in SDDS LH*

The original non-real-time SDDS LH* architecture consists of three basic software components:

1) A client that accesses data items in buckets,
2) A server which manages a single bucket located in RAM,
3) A split coordinator overseeing split operations of all buckets.

This type of the SDDS architecture does not support real-time data fetching requests. Such operations critical for real-time applications. Providing a real-time scheduling policy for read requests requires designing a proper internal structure of the server component. In our SDDS architecture we added support for handling real-time operations by using the priority queue and the requests scheduling algorithm in the server. Proposed server organization is shown in Figure 4.

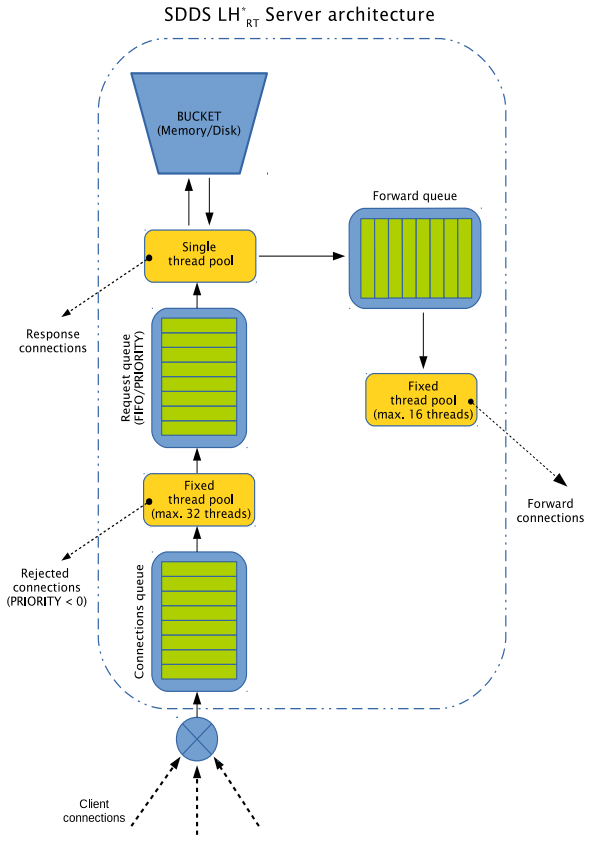


Fig.4. Server Organization

The incoming requests from clients are received by a pool of at most 32 threads that operate on a connection queue. The threads insert the requests into a request queue which is handled by a single thread. If a read request taken form the head of the queue meets its deadline, the thread fetches a corresponding data item from the bucket and sends it to the client. Otherwise the request is rejected. Requests that need to be forwarded to another servers are put into forward queue that is processed by a dedicated thread pool of at most 16 threads. In non-real-time SDDS LH* servers the request queue is a regular FIFO. In real-time servers a priority queue is used instead. Requests are ordered using Least Laxity First (LLF) scheduling method. Figure 5 shows the overall concept of LLF implementation used for SDDS LH*RT.

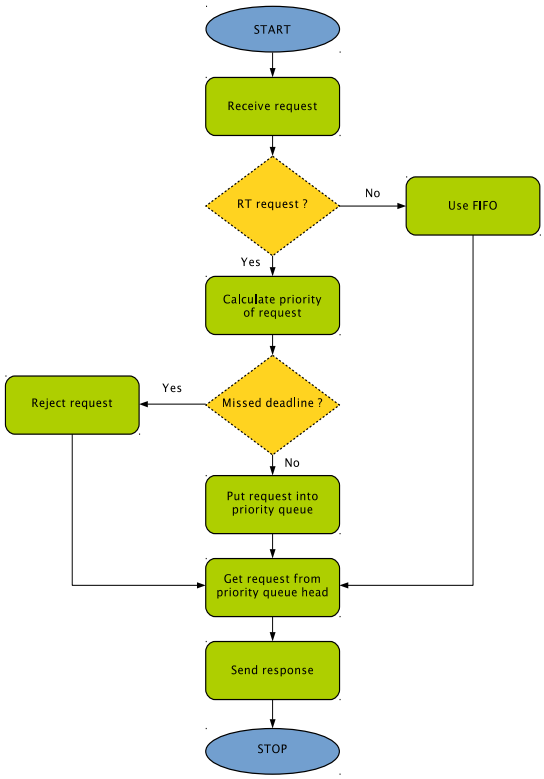


Fig.5. LLF Schema

After receiving a request the algorithm checks if it has real-time requirements. If no, the FIFO algorithm is applied for that request. Next, the LLF calculates a priority of the request. The priority is given by the Formula 1:

$$Priority = \begin{cases} \frac{1}{D - T_{rp}} & if\ D > T_{RP} \\ 0 & otherwise \end{cases} \quad (1)$$

where $D$ is a deadline and $T_{RP}$ is time of request processing. Requests are processed in descending order of their priorities. Requests with priority equal to 0 are rejected. The others are placed in appropriate locations of the priority queue, according to the value of their priorities. Then the algorithm processes a request from

 

the head of the priority queue and sends response to the client which made the request.

Since the time of accessing a data item in a bucket is negligible comparing to the average time of transmitting the item through a network ($T_{nm}$), we define the $T_{RP}$ as follows (Formula 2):

$$T_{RP} = n * T_{nm} \quad (2)$$

where the $n$ factor is equal 4 and it is introduced to provide for potential delays in network transmission. In the rest of the paper we refer to the real-time enabled SDDS LH* by the name SDDS LH*$_{RP}$.

## VI. Experimental Results

In the experiments we have evaluated the amount of rejected time-constrained read requests and the maximal queue length for the real-time and the non-real-time servers.

As an environment for tests we have used two nodes of a cluster computer. The client software was executed on a node with Intel Xeon E5410@2.33GHz processor (4 cores) and 16GiB RAM. Server software was run on a node with Intel Xeon E5205@1.86GHz processor (2 cores) and 6GiB RAM. Both nodes were connected through Gigabit Ethernet network.

In a single test scenario the client was sending a number of GET requests ranging from 100 to 10000. The deadlines for the request was chosen from 20ms to 2000ms accordingly to an estimated size of the request's response which ranged from 4KiB to 1024KiB. The stream of requests in each test was unordered i.e. the request formed a random pattern with the respect to the value of deadlines.

Figures 6 and 7 show that with the growing load, the SDDS LH* server (FIFO queue) may reject up to 97% of all requests, while the rejection rate of the SDDS LH*$_{RT}$ server (PRIORITY queue) stays below 13%. Thus, the QoS was improved by more than 7 times.

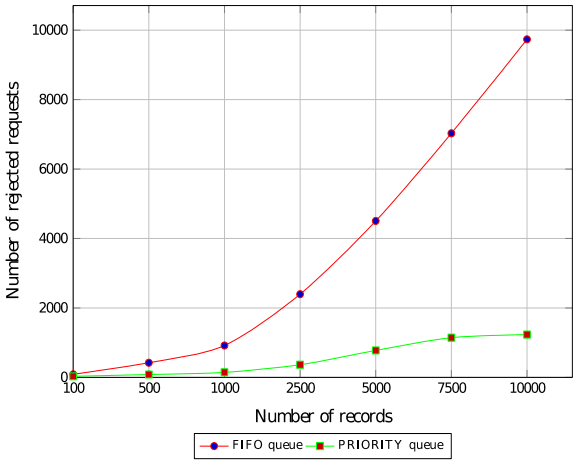


Fig.6. Rejected Read Requests with the Regard to the Scheduling Policy, Records Size 4kib - 1024kib, Deadlines 20ms - 2000ms

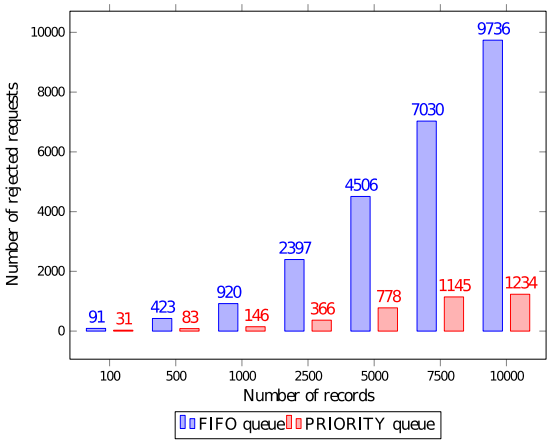


Fig.7. Rejected Read Requests with the Regard to the Scheduling Policy, Records Size 4kib - 1024kib, Deadlines 20ms - 2000ms

Above experiments showed that scheduling policy is very important in data stores with different times of processing the requests and/or different deadlines specified in requests. Real-time scheduling significantly increases the QoS but we may observe that the number of rejected requests grows slower with increasing load of the server.

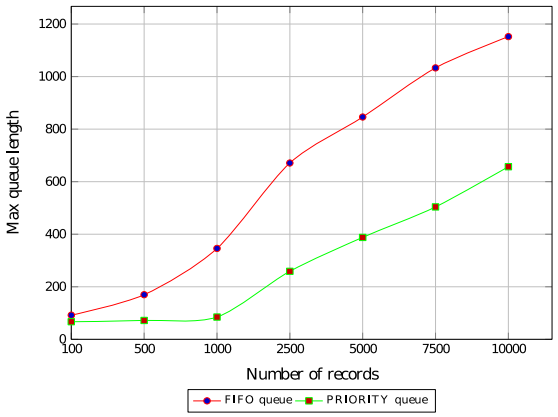


Fig.8. Maximal Length of the Requests Queue, Records Size 4kib - 1024kib, Deadlines 20ms - 2000ms

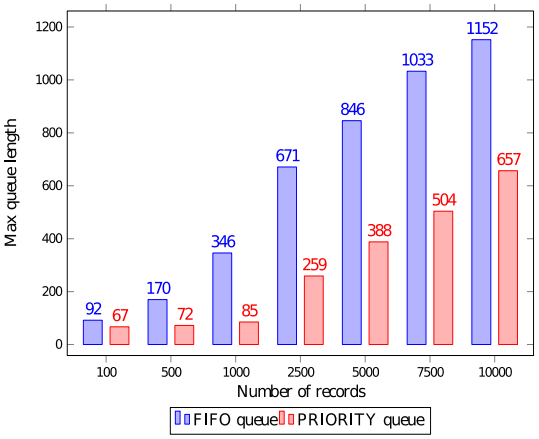


Fig.9. Maximal Length of the Requests Queue, Records Size 4kib - 1024kib, Deadlines 20ms - 2000ms

 

Plots 8 and 9 show the maximum length of requests queue for a given scheduling policy. With the increasing total number of request the SDDS LH*$_{RT}$ server unloads its request queue almost twice more efficiently than the SDDS LH* server.

Similar experiments were performed in the environment where one client was sending GET requests while other was sending UPDATE ones. Figures 10 and 11 present results obtained using the FIFO scheduling, while results obtained using real-time scheduling are given on Figures 12 and 13. For comparison, the results from the previous experiments are also presented. We may observe that in this case, significantly more requests were rejected in the PRIORITY queue while number of rejected requests processed in the FIFO order is almost unchanged. But still the real-time scheduling improves the QoS by 2 times in comparison with the FIFO scheduling.

UPDATE requests require more time for processing, thus such requests significantly decrease the performance of the data store. This results in higher number of rejected requests as well as a longer queues waiting for the processing. But still the queue length is shorter in the PRIORITY queue.

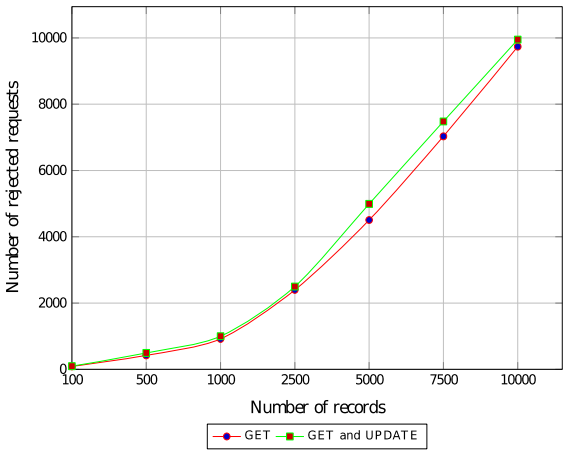


Fig.10. Number of Rejected GET and GET-UPDATE Requests for FIFO Queue

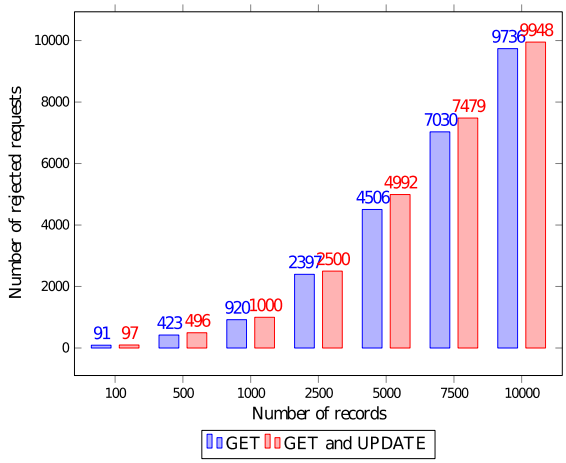


Fig.11. Number of Rejected GET and GET-UPDATE Requests for FIFO Queue

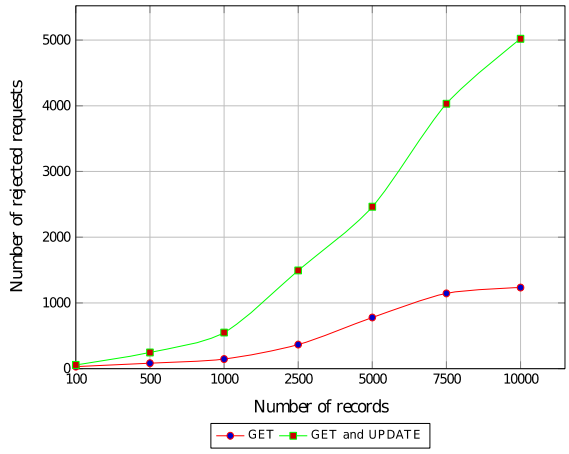


Fig.12. Number of Rejected GET and GET-UPDATE Requests for PRIORITY Queue

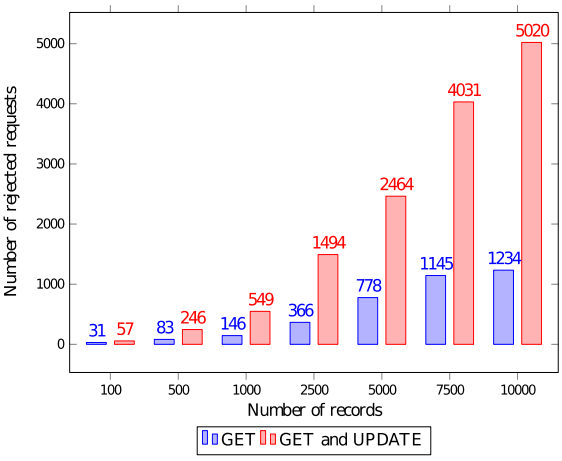


Fig.13. Number of Rejected GET and GET-UPDATE Requests for PRIORITY Queue

## VII. Conclusions

In this paper we have presented our work concerning real-time requests in the data store based on SDDSs. Instead of processing requests in the FIFO order, like in classical data stores, in our approach requests are scheduled using real-time scheduling policy. For this purpose the LLF algorithm was applied. The experimental results indicate that when a server is overloaded with incoming requests, even a relatively simple real-time scheduling policy allows it to process significantly more request, before their deadlines expire. Therefore, the number of rejected request is greatly reduced and the QoS is improved by more than 7 times.

Since our implementation was not optimized, we expect that the results could be further improved by applying aging of queued requests or using a dedicated implementation of the priority queue. In this way the time of scheduling will be reduced.

In our future work we plan to address the problem of assuring the firm real-time policy for data insertion requests and bucket split operations. Solving those problems will also require providing a real-time

 

forwarding of misaddressed requests. Finally, we expect to obtain a real-time distributed data store, that may be used in a wide range of real-time Internet applications.

## REFERENCES


[1] S. Goyal, "Public vs Private vs Hybrid vs Community - Cloud Computing: A Critical Review," *International Journal of Computer Network and Information Security(IJCNIS)*, MECS Publisher, IJCNIS Vol. 6, No. 3, February 2014, pp. 20 – 28.

[2] S. Bilgaiyan, S. Sagnika, S. Mishra, and M. Das, "Study of Task Scheduling in Cloud Computing Environment Using Soft Computing Algorithms," *International Journal of Modern Education and Computer Science (IJMECS)*, MECS Publisher, IJMECS Vol.7, No. 3, March 2015, pp. 32 – 38.

[3] G. Zhang and J. Liu, "The Study of Access Control for Service-Oriented Computing in Internet of Things," *International Journal of Wireless and Microwave Technologies (IJWMT)*, MECS Publisher, IJWMT Vol.2, No.3, June 2012, pp. 62 – 68.

[4] P. Hui, S. Chikkagoudar, D. Chavarría-Miranda and M. Johnston, "Towards a realtime cluster computing infrastructure," in *Real-Time Systems Symposium (RTSS 2011)*, The 32nd IEEE Real-Time Systems Symposium, Piscataway, NJ, IEEE (2011) 17-20.

[5] VoltDB, "Fast data-fast, smart, scale|voltdb," www.voltdb.com [Online: accessed 14-April-2015].

[6] B. Kao and H. Garcia-Molina, "An overview of real-time database systems," in *Advances in Real-Time Systems*, Springer-Verlag (1994) 463-486.

[7] S. A. Aldarmi, "Real-time database systems: Concepts and design," (1998).

[8] J. Lindström, "Real time database systems," in *Wiley Encyclopedia of Computer Science and Engineering*. (2008).

[9] D. Bigelow, S. Brandt, J. Bent, H. Chen, J. Nunez and M. Wingate, "Mahanaxar: Managing High-Bandwidth Real-Time Data Storage," https://systems.soe.ucsc.edu/node/389 [Online: accessed 14-April-2015].

[10] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, New York, NY, USA, ACM (2014) 157-168.

[11] T. Qian, A. Chakrabortty, F. Mueller and Y. Xin, "A real-time distributed storage system for multi-resolution virtual synchrophasor," in *PES General Meeting | Conference & Exposition*, 2014 IEEE , vol., no., pp.1-5, 27-31 July 2014.

[12] W. Litwin, M. A. Neimat, and D. A. Schneider, "LH* a scalable, distributed data structure," *ACM Transactions on Database Systems*, 21(4) (1996) 480-525.

[13] W. Litwin, M. A. Neimat, and D. A. Schneider, "RP*: A Family of Order Preserving Scalable Distributed Data Structures," in *Proceedings of the Twentieth International Conference on Very Large Databases*, Santiago, Chile (1994) 342-353.

[14] Y. Ndiaye, A. Diene, W. Litwin, and T. Risch, "AMOS-SDDS: A Scalable Distributed Data Manager for Windows Multicomputers," in *14th Intl. Conference on Parallel and Distributed Computing Systems - PDCS 2001*, (2001).

[15] K. Sapiecha, and G. Łukawski, "Scalable Distributed Two-Layer Data Structures (SD2DS)," IJDST (2013) 15-30.

[16] S. Deniziak, and S. Bak, "Synthesis of Real Time Distributed Applications for Cloud Computing," in *Computer Science and Information Systems (FedCSIS)*, 2014 Federated Conference on. (Sept 2014) 743-752.

[17] C. McGregor, "A cloud computing framework for real-time rural and remote service of critical care," in *Computer-Based Medical Systems (CBMS)*, 2011 24th International Symposium on. (June 2011) 1-6.

[18] W. Tsai, Q. Shao, X. Sun and J. Elston, "Real-Time Service-Oriented Cloud Computing," in *6th World Congress on Services*, SERVICES 2010, Miami, Florida, USA, July 5-10, 2010. (2010) 473-478.

[19] S. Liu, G. Quan, and S. Ren, "On-Line Scheduling of Real-Time Services for Cloud Computing," in *6th World Congress on Services*, SERVICES 2010, Miami, Florida, USA, July 5-10, 2010. (2010) 459-464.

[20] D. Kyriazis, A. Menychtas, K. Oberle, T. Voith, A. Lucent, M. Boniface, E. Oliveros, T. Cucinotta, and S. Berger, "A real-time service oriented infrastructure," in *Proc. Annual International Conference on Real-Time and Embedded Systems*.

[21] C. Freeny, "Automatic Stock Trading System," http://www.google.com/patents/ US6594643 (2003) US Patent 6, 594, 643.

[22] G. Fenu, and S. Surcis, "A cloud computing based real time financial system," in Bestak, R., 0002, L.G., Zaborovsky, V.S., Dini, C., eds.: ICN, IEEE Computer Society (2009) 374-379.

[23] O. Javed, Z. Rasheed, O. Alatas, and M. Shah, "KNIGHT$^{TM}$: a Real Time Surveillance System for Multiple and Non-Overlapping Cameras," in *Proceedings of the 2003 IEEE International Conference on Multimedia and Expo*, ICME 2003, 6-9 July 2003, Baltimore, MD, USA. (2003) 649-652.

[24] F. Lu, J. Wang, L. Cheng, M. Xu, M. Zhu, and G. K. Chang, "Millimeter-wave radioover-fiber access architecture for implementing real-time cloud computing service," in *CLEO: 2014*, Optical Society of America (2014) STu1J.1.


## Authors' Profiles

**Maciej Lasota** is a PhD candidate and Assistant Researcher at the Department of Computer Science in Kielce University of Technology, Poland. He received MSc in Computer Science from Kielce University of Technology in 2007. In 2008 he received Eng. in Control Engineering and Robotics also from Kielce University of Technology, Poland.

His primary technical and research interest is in distributed systems, with a particular focus on scalable distributed data structures, distributed data stores, cloud computing and real-time systems.

**Stanisław Deniziak** is Professor of Computer Science in Department of Computer Science, Kielce University of Technology, Poland. He received MSc in Computer Science from Warsaw University of Technology, and PhD degree from Gdańsk University of Technology. In 2006 he received DSc in Computer Science from

Warsaw University of Technology. Now, he is Vice Dean for Research and Promotion of Faculty of Electrical Engineering, Automatics and Computer Science, Kielce University of Technology.

He has published 85 research papers in various international and national journals, books and conferences. He is active reviewer end editorial member of 7 international journals such Journal of Systems and Software, Computing, Microprocessors and Microsystems, International Journal of Applied Mathematics and Computer Science, The Open Cybernetics & Systemic Journal, International Journal of the Physical Sciences, Annales UMCS - Sectio A Informatica. He has reviewed research papers of many international conferences like: IEEE Design Automation Conference, International Conference of Computational Methods in Sciences and Engineering etc.

Prof. Deniziak is IEEE and IEEE Computer Society Member.

**Arkadiusz Chrobot** is an Assistant Professor at the Department of Computer Science in Kielce University of Technology. He received MSc degree in Computer Science from Wrocław University of Technology in 2001 and PhD degree from Silesia University of Technology in Gliwice in 2012.

His research interests include distributed and real-time systems. Aside of research activities he is also involved in teaching of Basics of Programming, Operating Systems and Software Engineering.

**How to cite this paper:** Maciej Lasota, Stanisław Deniziak, Arkadiusz Chrobot,"An SDDS-Based Architecture for a Real-Time Data Store", International Journal of Information Engineering and Electronic Business(IJIEEB), Vol.8, No.1, pp.21-28, 2016. DOI: 10.5815/ijieeb.2016.01.03