A High Performance Enterprise Service Bus Platform for Complex Event Processing

Deng Bo Ding Kun Zhang Xiaoyi
Nanjing Telecommunication Technology Institute
Nanjing, China
E-mail: dengbomail@gmail.com, njdingkun@163.com, zxy31412@163.com

***Abstract*—Enterprise Service Bus (ESB) technology combines the Service-Oriented Architecture, which is based on the request/ response model, and the Event-Driven Architecture, which is based on the publish/subscribe model. It can satisfy the demand of loose couple communication and cooperation in the enterprise applications. However, existing ESB based systems can't process the complex events in the real-world applications very well. In this paper, we firstly give a complex event processing model based on the relational algebra, and then propose a complex event processing oriented enterprise service bus platform and a complex event stream processing engine, and give the main algorithms of complex event processing and their performance analysis. Experiments show that our platform performs much better than the existing ESB based systems in complex event processing.**

Keywords-Enterprise Service Bus; Service-Oriented Architecture; Event-Driven Architecture; complex event processing

I. INTRODUCTION

As an excellent architecture, Service Oriented Architecture (SOA) has been widely used in many fields, and leads the development of enterprise application software platform[1,2]. However, the request/response mode[3] used by SOA can hardly satisfy the requirement of loose couple communication and cooperation. Fortunately, the Event-Driven Architecture[4] can well meets such requirement, which is based on the publish/subscribe model[5]. As a result, the Enterprise Service Bus (ESB) comes into being against this background. ESB is a service computing platform based on event driven mechanism[6]. Its task is to offer an integration management platform to the Service Providers (SP) and the Service Consumers (SC), which decreases the coupling degree of SP and SC and eliminates the mess relationship between SP and SC. ESB manages the services of both the enterprise and its extension, and integrates the asynchronous messages transfer mechanism and the event driven model to satisfy the demand of enterprise applications.

However, the existing ESB systems (e.g. the mule system[7]) use the Message Oriented Middleware (MOM) and other mechanisms to implement event processing, including message storing and forwarding, priority control and so on, and can hardly process the complex events, such as filtering, aggregation and service choosing. On the other hand, the existing complex event processing technologies focus on event detection and communication[8-10], language describing and query optimization[11-13], and are short on consideration of how to implement the event stream processing engine in the service computing platform. Therefore, the complex event processing becomes a challenging technology in the ESB based enterprise application software platform.

This paper firstly gives the complex event processing model based on the relational algebra, and then proposes a complex event processing oriented enterprise service platform, designs the event stream processing engine, and gives the major algorithms of complex event processing and their performance analysis.

II. RELATIONAL ALGEBRA BASED COMPLEX EVENT PROCESSING MODEL

In enterprise applications, there are complex dependencies between various service providers and consumers, such as cooperation work, and any event of service providers or consumers is likely to affect the status of other service providers or consumers. Therefore, it is the duty for the enterprise applications software to integrate and process various related events. In this Section, we give the definitions of complex event firstly, and then present the complex event processing model based on the relational algebra.

Similar to [8], we give the definition of atomic event and event as follows.

Definition 1 (Atomic Event). The atomic event refers to an activity or a state changing. The atomic event $a = \{< id, v, t >\}$, in which id is the identifier of atomic event, v is an attribute or an attribute set, every element of v is an attribute value of a. For simplicity, we assume every atomic event has only one attribute, and the value$\in$ R (R is the real number domain), t is the time interval of the begin time and the end time of a, $t = [a_{start}, a_{end}]$.

Definition 2 (Event). Event $E = a_1 \cup a_2 \cup ... \cup a_m (m > 0)$.

Definition 3 (Complex Event). Let $D = \{E \mid E \text{ is event}\}$, the complex event $C = f(E_1, E_2, ..., E_n)$ $(n > 0)$, in which f is an arbitrary function on $D^n \to D$. We call f is an event construction function.

978-0-7695-3449-7/08 $25.00 © 2008 IEEE
DOI 10.1109/GCC.2008.66

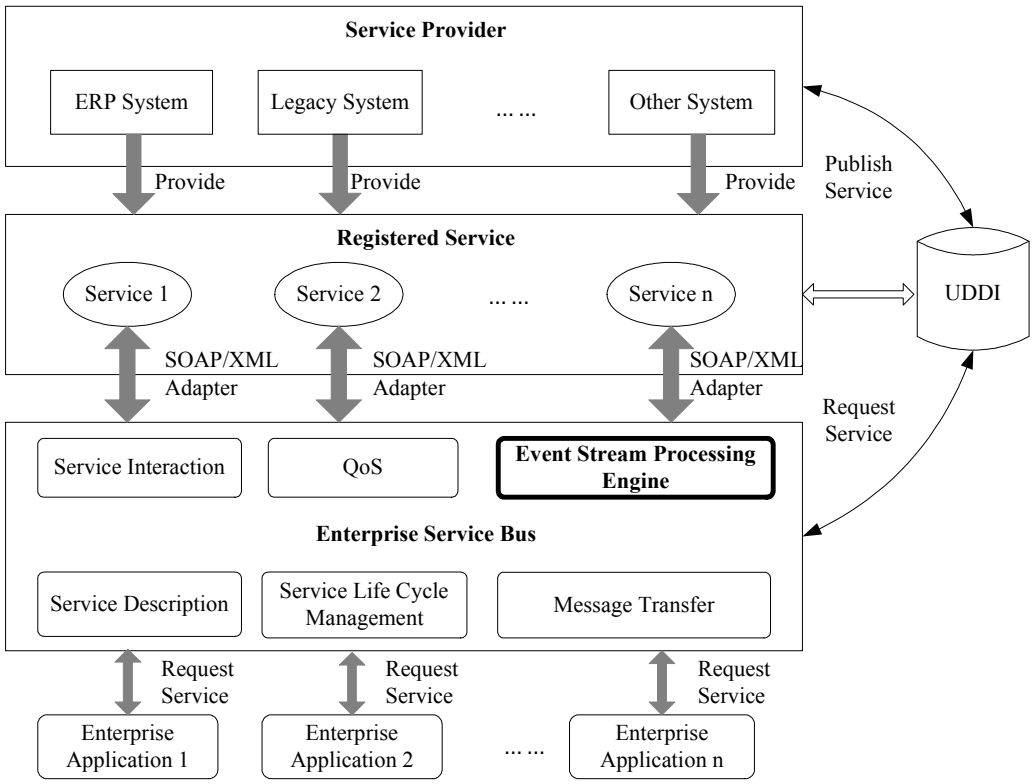

Figure 1. ESB Platform for Complex Event Processing

Clearly, there are various event construction functions, such as $\wedge$, $\vee$, etc. In practice, the complex event processing includes filtering, aggregation, service choosing, etc. In complex event processing, an atomic event can be dealt as a tuple with three fields, and we use the relational operations to implement selection, union, set-intersection, and set difference, etc.

Definition 4 (Event Selection Operation). Event selection operation σ is selecting the events that satisfy the predicate, the expression is $\sigma_p(E_1, E_2, ..., E_n) \quad (n > 0)$, in which p is the predicate.

To describe the time sequence, we use $t_1 \prec t_2$ to denote time t_1 before t_2, and use $t_1 \succ t_2$ to denote time t_1 after t_2.

Definition 5 (Event Aggregation Operation). Event aggregation operation G is computing the aggregation value of an event set during a specified period by an aggregation function. The expression is $G_{f,T}(E_1, E_2, ..., E_n)(n > 0)$, in which f is an aggregation function on $\mathrm{R}^n \rightarrow \mathrm{R}$ and satisfies the commutative law and the associative law, $T =< t_s, t_e >$, t_s is the start time and t_e is the end time. When computing the aggregation value, for arbitrary atomic event a in $E_1, E_2, ..., E_n$, if $a_{start} \prec t_s$ or $a_{end} \succ t_e$ then let the attribute value of a be 0, and v for otherwise.

Similar to the event selection operation and the event aggregation operation, we give the other event operations such as projection, selection, union, set-intersection, and set difference, etc. For the space reason, we will not go into details here.

In enterprise applications, when an event happens, a variety of services are needed to cooperate to process the event. The key to the cooperation is choosing the right services to process the right events at the right time. However, each service may have a number of instances (e.g. a variety of bank transfer services which provide the online transactions service), and the cost of service instance is different (e.g. the execution time), and what is more, the operation of the service instance may fail (e.g. transfer operation fails). Therefore, it is a challenge that how to choose the right service to minimize the cost of the event processing. As follows, we give the definition of the services choosing problem.

Let S be a service instance set, $S = \{s_1, s_2, ..., s_k\}$, $\forall s_i \in S$, let $cost_i$ denote the execution cost of s_i, and let pro_i denote the successful execution probability of s_i, for simplicity, we assume that pro_i of service instances are independent. Let W be an event processing service set, $W = \{w_1, w_2, ..., w_r\}$. $\forall w_i \in W$, at least one service instance satisfies the demand of w_i. Let sat_i denote the set of all service instances of S, which satisfy w_i. During the event processing, $\forall w_i \in W$,

the event processing system will choose and execute the service instances of sat_i one by one, until a service instance succeeds, or all service instances of sat_i fails.

Definition 6 (Service Choosing Problem). For an arbitrary event, the service choosing problem is how to choose an executing plan of the service instances of S to minimize the total cost of all event processing services of W .

The service choosing problem can be easily mapped to the shared filter ordering problem[14-16], and is NP-hard too.

III. ESB PLATFORM FOR COMPLEX EVENT PROCESSING

In this section, we propose a complex event processing oriented enterprise service bus platform, as is illustrated in Figure 1. This platform is based on the ESB architecture and uses an event stream engine to process complex events.

Figure 1 illustrates that ESB, as a mediator, connects the SPs and SCs and provides the services such as service interaction, life cycle management, Quality of Service (QoS), message processing and event processing, etc. In our platform, SPs register in the Universal Description Discovery and Integration (UDDI), and publish their services respectively. The published services individually connected to the ESB manages the services and sends messages between the SCs and the SPs. The SCs send their service requests to ESB service manager instead of sending to UDDI directly.

When dealing with service requests, our platform uses the asynchronous messages transfer mechanism and the event driven mechanism to assure that different information can be transferred with high accuracy, efficiency and security. Service request and response in the platform are transferred via SOAP/XML, and the SOAP and the XML standards are open and OS independent.

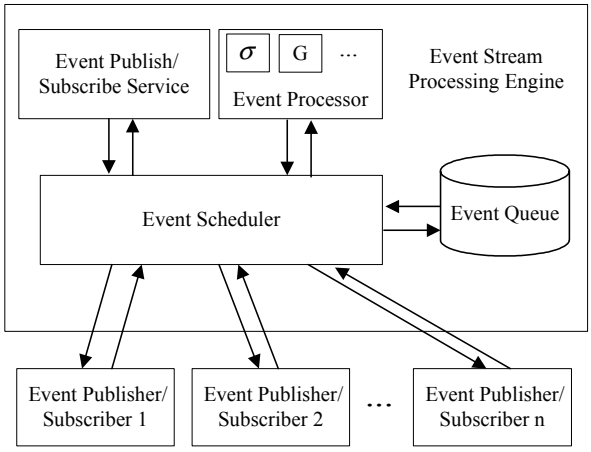

Figure 2. Event Stream Processing Engine

In our platform, we use an event stream engine rather than those popular technologies like MOM which simply forwards the events to process all kinds of complex events. The event stream engine is shown in Figure 2.

In the event stream engine, the event publisher (or subscriber) subscribes (or publishes) events through the publishing (or subscribing) service provided by the event stream engine. When event occurs, it is firstly stored in the event queue, and then processed by the event processor. After finishing the operations, such as selection (σ), aggregation (G) and so on, the event processor returns the results to event subscribers through event scheduler.

IV. MAIN COMPLEX EVENT PROCESSING ALGORITHMS

From previous section, we know that complex event processing algorithms mainly work in the event stream processing engine. In this section, we present some main complex event processing algorithms based on complex event processing model, including event selection, aggregation, and service choosing, etc.

A. Basic Complex Event Processing Algorithms

The basic complex event processing algorithms refer to the basic relational algebra operations of event, including event selection, projection, intersection, union, difference, join. The event selection algorithm is shown in Figure 3.

Algorithm 1. Event Selection Algorithm

INPUT: event stream $F_{input} = \{E_1, E_2, ..., E_n\}$, predicate p

0 $F_{output} \leftarrow \varnothing$

1 For each $E_i \in F_{input}$

2 If $p(E_i)$ then

3 $F_{output} \leftarrow E_i \cup F_{output}$

4 End If

5 Return F_{output}

Figure 3. Event Selection Algorithm

Let the average value of $| E_i | = \overline{m}$, each execution time of the predicate p be $O(p_t)$. In the best case, for each $E_i \in F_{input}$, the first input atomic event will produce the last result, i.e. $p(E_i)$ is $O(p_t)$. Therefore, the least complexity of the event selection algorithm is $O(p_t \cdot n)$. Assuming that in expected case, for each $E_i \in F_{input}$, half of atomic event are needed to produce the last result, then the average time complexity of the event selection algorithm is $O(p_t \cdot n \cdot \overline{m}/2)$. In the worst case, for each $E_i \in F_{input}$, all atomic events are involved in computing, therefore, the max_time complexity of the event selection algorithm is $O(p_t \cdot n \cdot \overline{m})$.

The other basic complex event operation algorithms are similar to the event selection algorithm. For the space reason, we will not go into details here.

B. Event Aggregation Algorithm

Now we describe the event aggregation algorithm. The event aggregation algorithm is mainly used in the statistical analysis of serial events to provide decision support for enterprises. The event aggregation algorithm is shown in Figure 4.

Algorithm 2. Event Aggregation Algorithm

INPUT: event stream $F_{input} = \{E_1, E_2, ..., E_n\}$, aggregation function f, start time t_s, end time t_e

0 $result \leftarrow 0$
1 For each $E_i \in F_{input}$
2 For each $a_{i,j} \in E_i$
3 If $\neg$((the start time of $a_{i,j} \succ t_e$) $\vee$ (the end time of $a_{i,j} \prec t_s$)) then
4 $result \leftarrow f(a_{i,j}) \cdot result$
5 End If
6 Return $result$

Figure 4. Event Aggregation Algorithm

As the aggregation function f meets commutative law and associative low, its correctness is easily to be proved. Let the average value of $|E_i| = \overline{m}$, and the execution time of the aggregation function f is $O(f_t)$, therefore, the time complexity of event aggregation algorithm is $O(\overline{m} \cdot f_t \cdot n)$.

In the event stream processing engine, when event aggregation algorithm finishes, the event processor generates a new event with the algorithm's return value as one of its attributes. Its start time is the earliest of all the atomic events in F_{input}, and its end time is the latest of all the atomic events in F_{input}. Thus the return value to event scheduler is an event, which is called aggregation event, rather than a value. And then event scheduler can send the aggregation events to subscribers.

C. Service Choosing Algorithm

In the loosely coupled environment, service choosing is one of the key steps in service cooperation. As described in Section Ⅱ, this problem can easily be mapped into the shared filter ordering problem, and is NP-hard too. It is also similar to the well-known set covering problem[17], but with the probabilistic nature (service instance could be success or failed with some probability), which makes more difficult to find sophisticated algorithms.

Now we give a service choosing algorithm based on greedy strategy. The core of this algorithm is to choose and execute a service instance which has a greatest "benefit/cost" ratio in all service instances. The "benefit" is the number of event processing services which can be satisfied by the service instance. The "cost" means the execution cost of this service instance. The greedy service choosing algorithm is shown in Figure 5.

Algorithm 3. Greedy Service Choosing Algorithm (GSC)

INPUT: event E_n, service instance set $S = \{s_1, s_2, ..., s_k\}$, event processing service set $W = \{w_1, w_2, ..., w_r\}$.

1 While $\exists$ unfinished event processing service do
2 For each unexecuted service instance s_i
3 $price_i \leftarrow 0$
4 For each unfinished event processing service w_j
5 If $s_i \in sat_i$ then
6 $price_i \leftarrow price_i + pro_i$
7 $degree \leftarrow 0$
8 For each unexecuted service instance s_v
9 If $s_v \in sat_i$ then
10 $degree$++
11 End If
12 $price_i \leftarrow price_i + (1 - pro_i) / degree$
13 End If
14 $price_i \leftarrow price_i / cost_i$
15 Select the service instance s_{max}, which has the max $price$, from all unexecuted service instances
16 Execute s_{max}
17 For each unfinished event processing service w_j
18 If $s_{max} \in sat_i$ then
19 Send result of s_{max} to w_j
20 End If

Figure 5. Greedy Service Choosing Algorithm

In the next section, we will prove that the performance of our platform is better than the existing ESB based systems in service choosing.

V. EXPERIMENT

In this Section, we first give the experimental setting in Section Ⅴ.A. In Section Ⅴ.B we experimentally compared the performance of our platform against the existing ESB based systems on complex event processing. The existing ESB systems lack the relational algebra based event stream processing engine, so we are not able to make comparison on event selection, projection and aggregation and so on. However, service choosing in event processing is necessary to every ESB based system. So we only give the results of service choosing of our platform and the existing ESB based systems in experiment.

A. Experiment Settings

The existing ESB systems always process events one by one in certain order. The greater the value of $pro_i / cost_i$ is, the higher priority the event gets in the sequence, i.e. if a event is of a lower cost and a higher probability of successful execution, it will be processed earlier. Such method is so called naïve service selection algorithm. Based on the naïve algorithm, we introduce a new algorithm called the Share algorithm, which

shares execution result of service instance for all associative event processing services, to compare against our GCS algorithm. In the following simulated environment, we compare GSC algorithm and Share Algorithm to verify the performance of our platform.

In the experiment, for simplicity, we take the execution time of service instance as the cost of it. Unless varied for a specific experiment, the parameters for the number of service instances is fixed at 100, the execution time of each service instance is initialized by the uniform distribution on [0.5, 5] seconds, the probability of successful execution is initialized by the uniform distribution on [0.8, 1], the number of event processing services is fixed at 200, $sat_i (1 \le i \le r)$ is initialized by the uniform distribution on [2, 8], i.e. on average there are 5 service instances for each event processing service satisfying the requirement of event processing, the mean of $sat_i (1 \le i \le r)$ is denoted as λ. Every experiment processes 1000 events and uses the total cost as the final result. The results go similarly with each other when the number of the processed events exceeds 200. For the space reason, we will not go into details here.

Our implementation of the test-bed and the related algorithms was written in GNU C++. All the experiments are conducted on Intel Pentium IV CPU 2.4GHz with 1GB RAM running Fedora Core 6.

B. Performance Compare

Figure 6 illustrates the comparison of performance while the number of service instances varying between 20 and 500. Figure 6 shows that our GSC algorithm behaves better and better than the Share algorithm does while the number of service instances grows. Once the number exceeds 50, the cost of our algorithm reduces to less than 75% of the cost of the Share algorithm. The main reason is that our algorithm is able to measure the benefit/cost ratio of each service instance more comprehensively and pick up more effective ones as the number of instances growing.

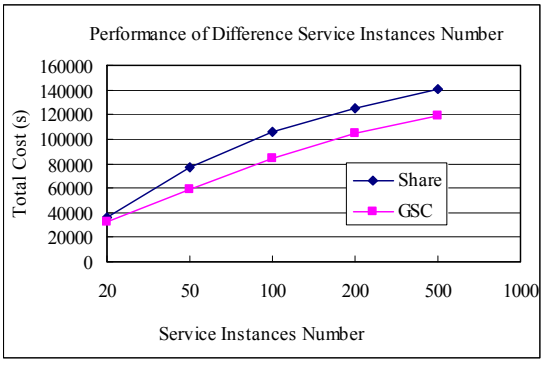

Figure 6. Performance of Difference Service Instances Number

Figure 7 illustrates the comparison of performance while the number of event processing services varying between 20 and 500. As is shown in Figure 7, our GSC algorithm behaves better and better than the Share algorithm does while the number of event processing services grows. Once the number exceeds 50, the cost of our algorithm reduces to less than 80% of the cost of the Share algorithm. The main reason is also that our algorithm is able to pick up more effective service instances as the number of event processing services growing.

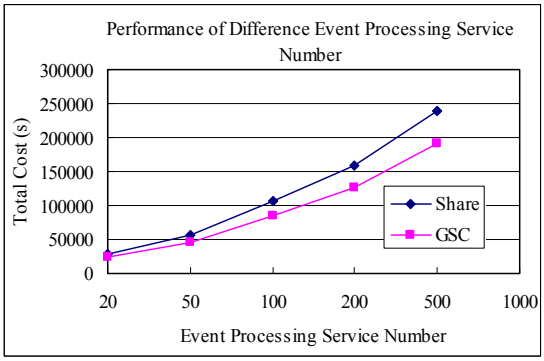

Figure 7. Performance of Difference Event Processing Service Number

Figure 8 illustrates the comparison of performance while the probability of successful execution of service instances varying. As is shown in Figure 8, our GSC algorithm behaves better and better than the Share algorithm does while the probability of successful execution grows. Once the probability exceeds 0.5, the cost of our algorithm reduces to less than 80% of the cost of the Share algorithm. That's because the effect of shared service instances goes better while the probability of successful execution grows, and our algorithm is able to pick up more effective service instances.

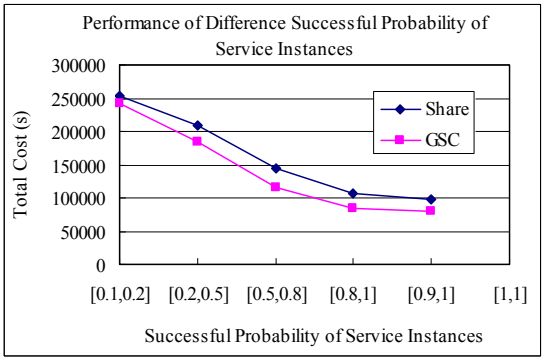

Figure 8. Performance of Difference Successful Probability of Service Instances

Figure 9 illustrates the results of performance comparison while λ (the mean of $sat_i (1 \le i \le r)$) varying. As is shown in Figure 9, when λ is no less than 3, our GSC algorithm behaves better and better than the Share algorithm does while λ decreases. Once λ is less than 10, the cost of our algorithm reduces to less than 70% of the cost of the Share algorithm. Because there will be more service instances need executing if λ decreases. As our algorithm is able to pick up more effective service instance at that time, the performance goes better.

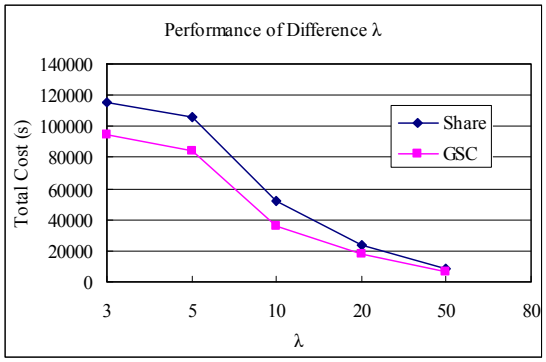

Figure 9. Performance of Difference λ

VI. CONCLUSION

It is the orientation of the enterprise application software platform to combining SOA and event driven architecture. The paper analyzes the deficiencies of the existing ESB systems in complex event processing, gives a new enterprise service bus platform for complex event processing, designs the event stream processing engine, gives the main algorithms of complex event processing and their performance analysis. The experiment result shows that this platform performs much better than the existing ESB based systems in choosing proper service to process complex events.

REFERENCES

[1] M.P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. Proceedings of the Fourth International Conference on Web Information Systems Engineering, Roma, Italy, 2003, pp. 3-12.

[2] P. T. Eugster, P. Felber, and R. Guerraoui et al. The many faces of publish/subscribe. ACM Journal of Computing, 2003, 35(2), pp. 114-131.

[3] D. Tombros, A. Geppert, and K.R. Dittrich. Semantics of Reactive Components in Event-Driven Workflow Execution. Proceedings of the 9th International Conference on Advanced Information Systems Engineering, Barcelona, Spain, 1997, pp. 409-422.

[4] Global Research partners. Event-Driven Architecture: the next big thing. Gartner Application Integration and Web Services Summit. August 2004.

[5] Y. Huang, D. Gannon. A Comparative Study of Web Services-based Event Notification Specifications. Proceedings of the 2006 International Conference Workshops on Parallel Processing, Columbus, USA, 2006, pp. 7-14.

[6] Sonic Software Corporation. Sonic ESB: an architecture and lifecycle definition. Sonic White Paper, 2005.

[7] Mule. http://mule.mulesource.org.

[8] J.H. Liu, Q.Y. Wu. An Event-driven service-oriented computing platform. Chinese journal of computers, 2008, 31(4), pp. 587-599.

[9] D.C. Luckham. The power of events: an introduction to complex event processing in distributed enterprise systems. Boston: Addison-Wesley, 2001.

[10] R.S. Barga et al. Consistent streaming through time: a vision for event stream processing. Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR). Asilomar, California, USA, 2007, pp. 363-374.

[11] E. Wu, Y. Diao, S. Rizvi. High-performance complex event processing over streams. Proceedings of the 2006 ACM SIGMOD international conference on Management of data, Chicago, USA, 2006, pp. 407-418.

[12] S. Rizvi. Complex event processing beyond active databases: Streams and uncertainties [Master Thesis]. UCB/EECS 2005-26, 205:46.

[13] A. Demers et al. Cayuga: A general purpose event monitoring system. Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR). Asilomar, California, USA, 2007, pp. 412-422.

[14] K. Munagala, U. Srivastava, and J. Widom. Optimization of continuous queries with shared expensive filters. Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Beijing, China, 2007, pp. 215–224.

[15] K. Munagala, U. Srivastava, and J. Widom. Optimization of continuous queries with shared expensive filters. Technical report, Stanford University, 2005. Available at http://dbpubs.stanford.edu/pub/ 2005-36.

[16] Z. Liu et al. Near-Optimal Algorithms for Shared Filter Evaluation in Data Stream Systems. Proceedings of the 2008 ACM SIGMOD international conference on Management of data, Vancouver, Canada, 2008, pp. 133-146.

[17] M. Garey and D. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman &Co., 1979.