Fault-Tolerant Complex Event Processing using Customizable State Machine-based Operators*

Thomas Heinze
SAP Research Dresden
Dresden, Germany
Thomas.Heinze@sap.com

Zbigniew Jerzak
SAP Research Dresden
Dresden, Germany
Zbigniew.Jerzak@sap.com

André Martin
TU Dresden
Dresden, Germany
Andre.Martin@tu-dresden.de

Lenar Yazdanov
TU Dresden
Dresden, Germany
lenar@se.inf.tu-dresden.de

Christof Fetzer
TU Dresden
Dresden, Germany
Christof.Fetzer@tu-dresden.de

ABSTRACT

Modern Complex Event Processing (CEP) systems often need an high degree of customization in order to implement required application logic. The use of declarative languages, such as CQL, often leads to complicated and hard to maintain application code. In this demo, we show how state machine-based CEP operators help to cope with these problems. State machine-based CEP operators allow for a high flexibility as well as a re-usability of application logic components. A major benefit of the presented solution is its easy integration with existing streaming engines, which we demonstrate using StreamMine, a highly parallel and fault-tolerant streaming engine prototype. In this demo we show: (1) how state machine-based operators allow for an easy definition of custom, reusable CEP operators, (2) how resulting state machines can be easily combined with existing fault-tolerance techniques within StreamMine and (3) how the resulting CEP applications can be tested in a cost efficient way.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications

General Terms

Algorithms, Design, Reliability

Keywords

Complex Event Processing

*This work is partially sponsored by European Commission's Seventh Framework Program (FP7) under grant agreement No. 257843

EDBT 2012, March 26–30, 2012, Berlin, Germany.

1. INTRODUCTION

Expressiveness and ease of use are perceived as being increasingly important factors which drive software purchasing decisions [8]. Moreover, many scenarios which require the use of Complex Event Processing are very domain specific. Many of these applications cannot be easily programmed using standard operators, and need extensive customization – the ability of standard eventing programming language, like CQL or CEL, to solve these tasks is very limited. Therefore, advanced concepts like ECA rules [6] or transactional logic [1] are often used.

In this demo we show case a solution based on user defined state machines which define CEP operators. The core idea of our system is to build an expressive and easy to use framework, which allows for operator reuse as well as extensibility, based on the users' needs and the scenario requirements. We demonstrate how user defined state machines can be easily integrated with existing streaming engines, using the StreamMine platform [5] as an example. Our approach (in contrast to other state machine approaches [3] which require user to work with streaming SQL dialects) allows users to create application logic using both state machine code and a visual RapidMiner interface.

Our work on StreamMine [5] as the underlying platform is motivated by its ability to process streaming data in a highly parallel fashion while ensuring fault-tolerant execution of application logic. This is dictated by the fact that the complexity and the amount of data in modern CEP scenarios is rapidly increasing: OPRA estimates that in 2013 one should expect over 10 million transaction events per second originating from Securities Industry Automation Corporation [2]. Therefore, in order to cope with such workloads highly parallel and distributed processing infrastructures must be used. However, with the increasing number of nodes in CEP systems, the probability of failures increases as well [9]. This in turn implies that the imminent node failures must be handled with minimum overhead and precise recovery of the system state. The precise recovery of the system state is a key feature of StreamMine, which we show in this demo. State recovery is especially crucial as such state is usually accumulated over long periods of time.

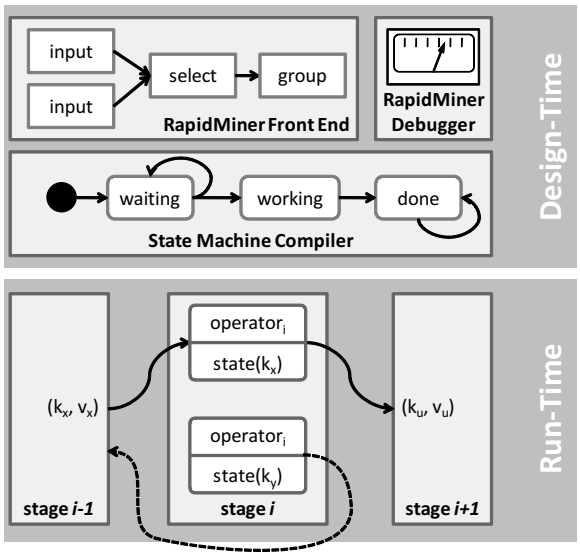

Figure 1: Stage-based architecture of StreamMine

2. SYSTEM ARCHITECTURE

The StreamMine prototype shown in this demo consists of runtime and design time components – see Figure 1. The design time contains a user interface which allows for a GUI-based query definition, as well as online monitoring of the application logic execution in the underlying streaming system. The StreamMine runtime processes the event stream as event batches and allows the users to define the size of batches (down to the size of one event), trading off throughput for latency. The processing logic is divided into stages with each stage processing a full batch before sending it to the next stage.

In order to process data in parallel StreamMine uses a key-based stream partitioning approach. A single stream is split across several slices (within one stage) which execute the same stage logic in parallel. The number of slices per stage can be arbitrarily chosen, which allows for flexible resource allocation driven by the complexity of the stages' operators.

3. ACTIVE REPLICATION

Due to the high event rates and infinite event streams observed in typical CEP scenarios, logging of all events to allow for recovery in case of crashes would introduce an unacceptable overhead. Therefore, within this demonstration we will show how an enhanced version of active replication [4] can be used to mitigate this problem. Our variant of active replication does not require a continuous duplication of workload across identical nodes. Instead we allow nodes to use idle CPU time to provide fault-tolerance by acting as an active replica for another node.

Figure 2 illustrates the scheme used for our active replication. For each slice a primary and a secondary (backup) copy is created. In case of normal utilization (below 50%) the node responsible for secondary copies performs same computations as the primary copy with the successor stage ignoring the duplicated input. If the primary copy fails, the secondary copy becomes the primary and takes over the processing with no latency.

In case when utilization of the node hosting a secondary

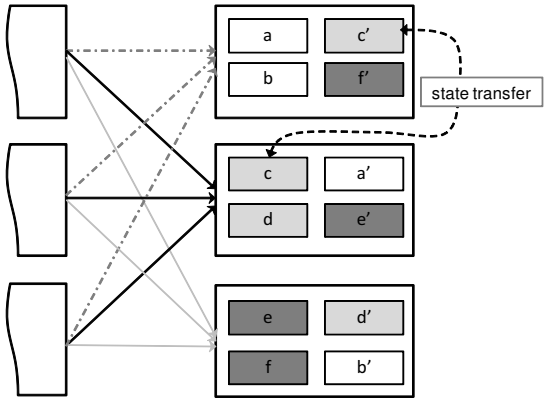

Figure 2: Combining active and passive replication in StreamMine

copy exceeds 50% it automatically switches from active replication to passive replication. In passive replication mode, the secondary node buffers all incoming events, without processing them. This allows it to restore the state in case of the primary failure. When the event buffer of the secondary is full, it triggers the state transfer from the primary and discards the events from its buffer. Events can be discarded as the corresponding state has already been installed in the secondary. The primary and secondary slices are distributed across all slices of a single stage in such a way that the processing is equally partitioned over all available nodes.

A cautious reader might notice, that to enable the above active/passive replication scheme one needs to be able to perform deterministic execution of the application logic. Deterministic execution allows for replaying of the events with consistent results. StreamMine replication scheme ensures that state of the different replicas is consists at any time. Based on our previous work [5], we demonstrate that using active replication introduces only a small overhead for the processing, in terms of both throughput and latency.

4. STATE MACHINE-BASED OPERATORS

In order to represent the processing logic within one stage a state machine is used. The state machine encodes the internal state of the operator as well as the handling of arriving events. We choose the state machine approach because we believe it is well suited to model complex behavior and allows for modularized as well as extensible design of operator semantics.

Our state machines consist of states and transitions, which describe the transfer between states. A transition can be watched by a guard, which ensures that given conditions are met before the transition is taken. In addition, transitions can trigger specific actions, such as sending of an event or increasing a counter.

Each time a new event arrives a transition inside the state machine is called. This, in turn, allows other transitions to become active resulting in further state transition. This is repeated until the state machine does not perform any state transition or the state machine arrives in its initial state again. Because we allow state machines to be defined by the user, it can potentially happen that an infinite cycle of states is created. A simple solution to this problem, based on detection of strongly connected components, has been

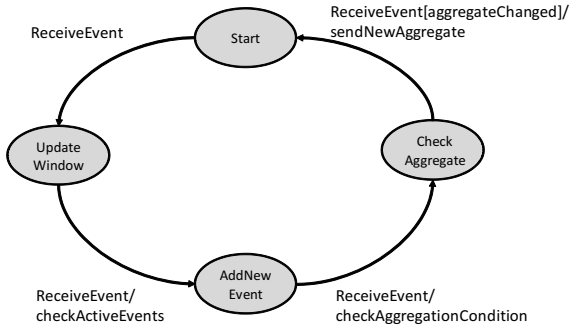

Figure 3: Example state machine-based windowed aggregation operator

shown in [11]. Solution presented in [11] has the complexity of $O(n)$, where n is the number of states.

Figure 3 illustrates an example state machine which implements a windowed aggregation function. The processing of new input event always begins in the start state and consists of three major steps: (1) removing all outdated events from the time window, (2) adding the new event to the aggregate and (3) sending the new aggregate to the next stage. All these steps are modeled as a single state and by taking the transition between the different states corresponding actions are triggered, in order to remove outdated events or to send the new aggregates to the next stage.

To integrate state machines into StreamMine we use an already existing approach called State Machine Compiler (SMC) [7], which offers a domain specific language (DSL) to represent state machines and allows for an automatic compilation into native code. From within the generated code external classes and/or functions can be accessed, which is used (among others) to send events from one StreamMine stage to its successor stages. The DSL representation of the state machine shown in Figure 3 is shown in Figure 4. Figure 4 also highlights the different components of the state machine.

As a starting point for our system we have created a set of state machine templates representing commonly used CEP operators including: selection, join, aggregation, projection and sequence. These templates can be instantiated with parameters to allow for an easy reuse by, e.g., setting filter criteria or join conditions. In addition, these predefined solutions can serve as starting point for user defined operator semantics, because the implemented actions and guards can used also in other state machines.

	State Machine DSL	Generated Code
Lines of code	304	2087
Ratio	1x	6.8x

Table 1: Comparison of the number of lines of code written for the state machine DSL and generated by the SMC compiler

We have evaluated our approach with the help of the 2011 DEBS Challenge scenario [10]. The 2011 DEBS Challenge scenario is a good candidate for evaluation as it requires a

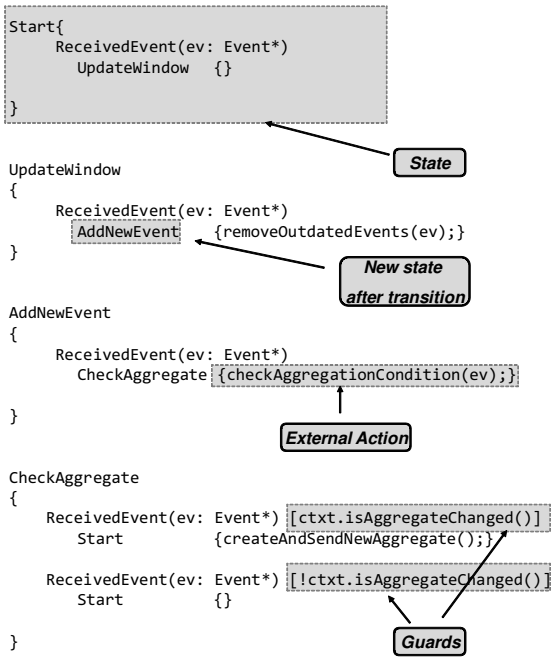

Figure 4: DSL-based implementation of the state machine-based windowed aggregation operator shown in Figure 3

large number of custom-written operators. Table 1 shows a comparison between the number of lines of code written using the state machine DSL and the number of code lines generated by the SMC compiler for the Challenge solution. Using the abstraction offered by the state machine-based operators only a small fraction of code needs to be implemented by hand.

5. FAULT-TOLERANT OPERATORS

Besides being well suited for describing complex operator semantics state machines-based operators can easily be integrated into the existing parallelized and fault-tolerant StreamMine platform. The execution of state machines can be parallelized by instantiating several identical state machines and executing all of them on different parts of the input stream.

To evaluate the performance of the state machine-based operators we have measured throughput and latency – see Figure 5. Evaluation was performed on a 50-node cluster with each node equipped with 2 Intel Xeon E5405 (quad core) CPUs and 8 GB of RAM. Conducted experiments demonstrate that using 12 instances in parallel a throughput of up to 250,000 events per second can be achieved. In addition, the end to end latency per stage is below 3 milliseconds. It can be concluded, that by using only the state machine-based operators one can build scalable applications.

The state machine operators are also well suited for integration with the existing active replication solutions. An important aspect of implementing active replication inside a streaming engine is that the state of an operator which has to be synchronized between different replicas has to be

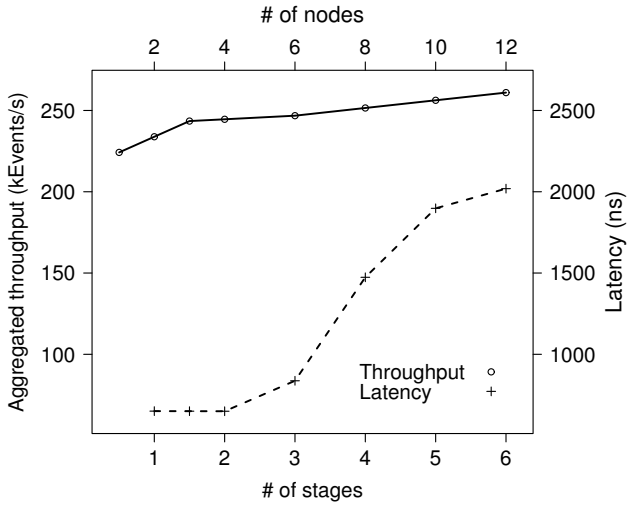

Figure 5: Throughput and latency using the state machine-based operators

specified. When allowing users to write custom application logic, they also need to manually specify which data has to be synchronized between the replicas. In contrast, using the state machine-based operators this can be done automatically, as the application logic state is stored in the current state of the state machine.

6. OFFLINE TESTING

To simplify the usage of the state machines we created a graphical representation of the application logic allowing for easy creation and composition of state machine-based operators. The use of RapidMiner framework in combination with StreamMine allows users to conduct offline debugging of their query graph logic using a subset of input data.

Application logic created using RapidMiner user interface can be compiled and executed locally thus allowing cost neutral testing and debugging, especially if the target environment implies the need to purchase expensive on-demand infrastructure cycles. To that end we use a small scale version of the StreamMine engine which is integrated with the RapidMiner user interface. Operators written using the state machine DSL can be automatically recognized and imported into the RapidMiner interface. From there the generation of the local debug code can be undertaken. While offline debugging using the RapidMiner does not replace the online debugging on a real system, in our opinion, it presents a cost effective and a fast alternative to testing using target environments.

7. DEMONSTRATION SETUP

In the following we briefly describe the demonstration setup. The demo consists of two major parts: one showing the usage of the user defined operators and the second illustrating their combination with active replication techniques.

7.1 User Defined Operators

This part of our demonstration will introduce the user interface based on RapidMiner and demonstrate the integration between the user interface and the underlying StreamMine system. We will show how complex operator semantics can be implemented fast and in an intuitive way using the RapidMiner user interface. In addition, we will showcase the deployment of a prototypical application and demonstrate live monitoring of the overall system performance.

7.2 Fault-Tolerance Mechanism

During the second part of the demo we will focus on presenting the limited overhead introduced by our active/passive replication-based fault-tolerance. We will show the functioning of the active replication in combination with the state machine-based operators. We will shown, that the correctness of the results can be ensured as long as at least one replica is running. To this end we will kill replicas simulating crash-stop failures. In addition, we will show how the system can react to sudden peaks in the event load and automatically switches to passive replication.

8. REFERENCES

[1] D. Anicic, P. Fodor, R. Stühmer, and N. Stojanovic. Event-driven approach for logic-based complex event processing. In *2009 International Conference on Computational Science and Engineering*, pages 56–63. IEEE, 2009.

[2] J. P. Corrigan. OPRA updated traffic projections for 2012 and 2013. Technical report, Options Price Reporting Authority, August 2011.

[3] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, W. White, et al. Cayuga: A general purpose event monitoring system. In *Proc. CIDR*. Citeseer, 2007.

[4] A. Martin, C. Fetzer, and A. Brito. Active replication at (almost) no cost. In *Reliable Distributed Systems, 30th IEEE Symposium on*. IEEE, 2011.

[5] A. Martin, T. Knauth, S. Creutz, D. B. de Brum, S. Weigert, A. Brito, and C. Fetzer. Low-overhead fault tolerance for high-throughput data processing systems. In *Distributed Computing Systems, 31st International Conference on*, Los Alamitos, CA, USA, June 2011. IEEE Computer Society.

[6] A. Paschke, A. Kozlenkov, and H. Boley. A homogeneous reaction rule language for complex event processing. 2010.

[7] C. Rapp. The state machine compiler (SMC).

[8] R. L. Sallam. BI platforms user survey, 2011: Customers rate their BI platform functionality. Gartner Research Note G00211770, March 2011.

[9] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 193–204, New York, NY, USA, 2009. ACM.

[10] N. Stojanovic. DEBS challenge. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pages 369–370. ACM, 2011.

[11] R. Tarjan. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 114–121. IEEE, 1972.