DEBS'11 Grand Challenge: Streams, Rules, or a Custom Solution?

Lynn Aders, René Buffat, Zaheer Chothia, Matthias Wetter,
Cagri Balkesen, Peter M. Fischer, Nesime Tatbul
Department of Computer Science, ETH Zurich, Switzerland

Technical Report #749
ETH Zurich, Department of Computer Science
July 2011

DEBS'11 Grand Challenge: Streams, Rules, or a Custom Solution?

Lynn Aders, René Buffat, Zaheer Chothia, Matthias Wetter
Cagri Balkesen, Peter M. Fischer, Nesime Tatbul
Department of Computer Science, ETH Zurich, Switzerland
{laders, rbuffat, zchothia, wetterma}@student.ethz.ch
{cagri.balkesen, peter.fischer, tatbul}@inf.ethz.ch

ABSTRACT

This paper describes how we modeled and solved the DEBS'11 Grand Challenge of implementing a social network game using event processing technology. We first present an automaton-based model that we used to capture the game semantics. Then we summarize three different approaches we investigated to implement this automaton together with their evaluations. Finally, we provide a discussion of our observations and lessons learned as a result of this study.

1. INTRODUCTION

The fifth ACM International Conference on Distributed Event-Based Systems (DEBS'11) has defined a grand challenge for implementing an event processing application [1]. The application involves Trivia Geeks Club, a game that is played within a social network group. Players receive a periodic stream of questions and post answers. The rules of the game define how questions should be answered and how answers should be scored. At the heart of the application is a complex scoring system, which not only assigns scores simply to correct answers but also to other additional situations such as best daily score or 10 consecutive correct answers. As such, the challenge includes a set of event producers, event consumers, a scoring system, and a few other system functions, which need to be implemented using event processing technology, but without the use of a database system.

We participated in the challenge as a team of four students and three mentors from ETH Zurich. Our approach to solving the challenge was to study three alternative ways by implementing each and then comparing. More specifically, we implemented the game based on a custom solution, a rule engine, and a stream processing engine. All of these implementations are commonly based on a formal, automaton-based model that we constructed, according to our own interpretation of the high-level informal specification of the challenge given at the web site [1]. We have also implemented a test case generator to validate the correctness of our implementations against this automaton-based model.

In this short paper, we describe our solution approach to the DEBS'11 Grand Challenge and our findings. We start with our modeling of the problem in Section 2. Then we present the three alternative implementations in Section 3. We finally conclude in Section 4 by discussing our observations and findings about how the three solutions compare along several key dimensions.

2. MODELING THE PROBLEM

The problem posed for the challenge is to implement a simple game, where users of a social network participate in answering trivia questions throughout the day. Each question is valid for 30 seconds and the user can choose from among four given answers. In addition, the user can annul their answer within 10 seconds or inquire about the most-frequent answer (MFA) among all participants. There is a set of rules which determine the number of points a user receives. Besides checking an answer for correctness, one would also like to determine, for example, the first user to correctly answer a question or the user with the highest daily score. The objective of the challenge is to implement this game using event-processing systems, where one processes a data stream on the fly, without using a database. As an additional challenge, the processing system should support late arrival of answers, with a bound of at most two seconds.

Based on the high-level informal specification of the challenge [1], we first tried to formally model the problem by clarifying some edge cases as well as issues about time handling. As a result, we modeled the scoring rules in the form of a state machine, as depicted in Figure 1. This represents the interaction with one user over a single question. State transitions are triggered by arrival of a new event (e.g., MFA request) or after a predefined time (e.g., finalize an answer once no further annulments are possible). This automaton is specific to an individual user or question. We were able to use the output of this automaton to calculate all further scoring rules. A second automaton is responsible for awarding a bonus for three consecutive correct answers and a penalty in case of ten incorrect answers in a row. However, the scoring rules for the daily, weekly, monthly, and most appearances in the top5 bonus need global knowledge. All of this rules can be calculated by aggregate over all users: computing the best score over a given time frame (day, week, month) requires no additional state, but rather just accumulating the output of each automaton. The first correct answer score can be assigned to the user with the smallest timestamp of a correctly answered event per question.

3. OUR SOLUTION APPROACH

In this section, we describe the three alternative approaches that we followed to solve the DEBS'11 Challenge. The first is built from scratch with no dependencies, the latter two are based on Complex Event Processing (CEP) systems: *Drools* [2], a rule engine and *Borealis* [3], a stream-processing engine.

3.1 Custom Solution

3.1.1 *Overview*

One avenue we pursued was to build a system from scratch without use of any framework. This solution corresponds closely to the formal model described in the previous section and was implemented with Python [7]. This choice was due to the expressiveness of the language, which comes at the expense of run-time perfor-

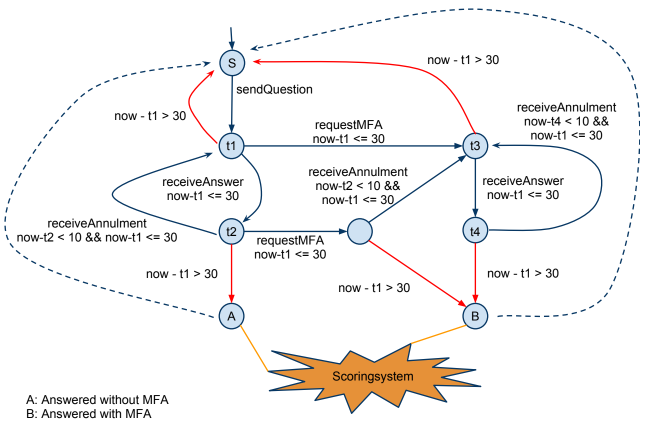

Figure 1: State machine for assigning scores to a single user

mance.

3.1.2 *Implementation*

The implementation is a direct translation of the state machine presented in Section 2, with state stored explicitly per user. There are functions to handle each input event, which trigger the corresponding state transitions. Time-based transitions are implemented by means of callbacks, which are scheduled according to a virtual clock driven by the stream time. Input events refer to users by their IDs. A hash table is used to support efficient lookup of the corresponding user state.

The custom solution includes several improvements over the formalization given previously. For example, with the state machine in Figure 1, on receipt of an answer one would need to wait for the annulment period to elapse before finalizing an answer. Instead, we take an eager approach and award points immediately and only reverse this if an annulment is received. Points are awarded to an internal scoreboard, which is flushed at the end of each question. This is necessary so as not to expose intermediate state. By processing early, the event processing load is spread more evenly over the duration of the question. A further improvement is to track the number of consecutive correct and incorrect answers not by an automaton as previously mentioned, but rather with two integer counters, which mutually reset each other. This enables one to change the rule itself at run time (e.g., 5 consecutive answers instead of 3).

3.1.3 *Evaluation*

Compared with the other solutions, this solution was easy to build, as there were no interfaces to understand or interact with. We had a first working version within about a week of development. This owes partially to the use of a dynamically typed language, which facilitates rapid prototyping. Whilst Python was a good choice for this setting, it may not be appropriate for a project with larger scale and several developers. Further, implementation effort for a custom solution is highly dependent on the desired feature set. For example, support for distribution or fault tolerance are not available and would need to be incorporated separately.

3.2 Rule-based Solution

3.2.1 *Overview*

As a second approach, we implemented the game using a rule engine. A rule engine consists of a rule base, a working memory containing facts, and an inference engine that reasons based on the rules over the collection of facts. It determines whether the facts in the working memory fulfill the conditions of a rule. If so, the action part of that rule is executed.

We decided to use Drools [2] as our rule engine mainly because it has a sophisticated concept of event processing. Drools introduces an event as a special case of a fact that has a notion of time. Thus, one can model rules on the occurrence or absence of events. Drools provides a set of temporal operators which allow one to correlate events in time. For example, one can define the constraint that an event A has to occur between 0 and 30 minutes after another event B. Drools also supports garbage collection of events, where events which are no longer needed as they can no longer match any rules are automatically detected and removed from working memory. In our previous example, event B could be removed after 30 minutes. Furthermore, Drools supports both time-based and size-based sliding windows.

3.2.2 *Implementation*

For implementing the challenge, we made use of two components of Drools: (i) *Expert*, which provides pattern matching over Java objects via the RETE algorithm [5], and (ii) *Fusion*, which adds stream processing functionality. First of all, we defined the rule base that contains the description of the game rules. Then, we wrote a small number of Java classes to receive the events from the source, insert them into the rule engine, and model the events and facts themselves. For example, we have a class `User` and a class `UserMap` to represent a collection of Users. Both are accessed as facts in the working memory. The `User` object includes fields to save states like the actual score or the number of questions that were answered correctly in a sequence. Using methods of the `UserMap` class one can for example calculate the user with the best daily score. These methods can be invoked via rules that are triggered by temporal constraints (cron jobs).

```
rule "correct answer"
when
  $u: User(userId == $userId)
  $q: Question(
    $questionId:questionId,
    $correctAnswer:correctAnswer)
    from entry-point "EventStream"
  $a : Answer(
    $userId:userId,
    $answer:answer,
    questionId == $questionId,
    answer == $correctAnswer,
    this after[0s,30s] $q)
    from entry-point "EventStream"
  not(AnswerAnnulment(
    this after[0s,10s] $a,
    questionId == $questionId,
    userId == $userId)
    from entry-point "EventStream")
then
  entryPoints["ScoreStream"].insert(
    new UpdateScore($userId, $questionId));
  modify($u) {
    reportCorrectAnswer($questionId);
    resetIncorrectAnswersCount();
  }
end
```

Listing 1: Rule for "correct answer"

In Drools, rules are declaratively expressed in the form of two clauses: (i) `when` clause contains the constraints that determine when and if a rule matches, and (ii) `then` clause defines the action to execute when the rule fires. As an example, Listing 1 shows the rule that checks if an answer is correct. In the example, specific types and variables are bound via pattern matching. Furthermore, time relations are utilized to enforce that the answer is valid (i.e., it is within the 30 second question time frame) (see line 13) and it has not been annulled (see line 16). In the `then` clause, we first create a new event to distribute points for the correct answer (`UpdateScore`), and then we mutate the user fact via the `modify` statement, which allows Drools to reason again over other rules involving the user, which may now be applicable.

In total, we wrote 19 rules, of which 10 correspond almost one-to-one with the rules given in the challenge specification; 3 additional rules help implement the logic to handle annulments; and the remaining 6 are used to handle the MFA requests, to output the score updates, or to explicitly retract events out of the working memory.

3.2.3 Evaluation

The Drools framework provides several tools to ease development: temporal reasoning, pattern matching, and automatic life time management of events. Overall, we have the experience that rules are an elegant manner to express business logic. They are well-encapsulated and easy to read. A further advantage of Drools is that data is decoupled from the relevant logic, which allows one to easily adapt to changing rules.

Unfortunately, not everything behaved as expected. Regarding automatic memory management, we observed that, for example, answer events were retained in memory beyond the duration of a question, which led to exhaustion of heap memory. An attempt to resolve this problem by explicitly specifying lifetimes for events was not successful. The final workaround was to introduce a new rule that explicitly kills an event object after a certain time period. We also found reasoning about performance of rule matching constructs to be challenging, given that a single construct may express complex logic. Furthermore, we were unable to find a profiler which could be applied to the rules. Lastly, we were not able to make use of time frame windows due to performance reasons, instead of which we used pure Java code as Drools allows using custom Java code in combination with the rule engine.

3.3 Stream-based Solution

3.3.1 Overview

As a third alternative, we implemented the challenge using a stream processing engine (SPE). In contrast to database systems where queries are executed over stored data, in SPEs, data is streamed through stored queries.

We decided to use Borealis [3] as our streaming engine, not only because it is open-source and supports distributed operation, but also some of us already had previous experience with Borealis. Borealis is a tuple-driven SPE [4], which supports distribution over multiple machines by allowing the division into so-called processing nodes. Each such node processes a set of streams, each of which is an append-only sequence of tuples consisting of a well-defined set of attributes. On each node, a set of operators processes one or more streams and routes the results to further nodes. An operator can be stateless, meaning that it performs operations on single tuples without keeping any information from one tuple to the next (e.g., *filter*, *map*, *union* in Borealis). Operators can also be stateful, meaning that each execution acts over a well-defined set of tuples called a "window' (e.g., *aggregate*, *join* in Borealis). Furthermore, the user has the possibility to add custom operators to the Borealis default operator set by implementing a given set of functions.

3.3.2 Implementation

In our implementation, we were able to easily use several features of Borealis. Defining the query network of operators that implements the game rules and scoring is done through an XML file in a pretty straightforward way. The handling of the input streams and how the tuples are delivered across operators is done internally. Predefined functions can be adapted to read and parse the input from a file as well as writing the output to the console or to a file. The windows for the stateful operators can be based on the number of tuples or on a time period. Beyond the existing operators, the possibility of implementing custom operators allows to fulfill the tasks for which the provided functionality is not sufficient.

Figure 2 illustrates our query network that implements the game. We created a custom operator (`Local_score`) to process all rules requiring knowledge about only one user. The rules for `FirstWhoAnswered` and `ComputeMostFrequentAnswer` are also implemented as separate custom operators. Finally, a fourth custom operator (`Global_score`) processes all rules requiring only the daily sum of each user. With these four custom operators, we were able to implement the main functionality of the system according to our detailed specification. One additional custom operator was needed in order to allow a change in the number of received points for a certain bonus and another to pre-process the data to manage out of order tuples. This design enables us to distribute the query network with ease. Also, all operators save resources by only forwarding the information needed at a later stage. For instance, only the daily sum of each user has to be sent out to the global operator.

In addition to the custom operators, we have also added two mechanisms into Borealis for ensuring low-latency results. First, in a distributed setup with several operators processing individual users, it is not guaranteed that all operators receive tuples on each question. To avoid possibly high latencies due to window timeouts, we introduced punctuations into Borealis [6], which signal to the operators that the current question has finished. This allows the operators to timeout all pending users. Second, we also introduced a synchronization stream that forwards how many users are active per day to the operator processing the global rules. The global operator lacks this information without the synchronization stream, because it has only local knowledge and can therefore only timeout if tuples from the next day arrive (which would introduce a latency of 24 hours for global rules into our system).

3.3.3 Evaluation

As described above, to implement the challenge, we created 4 custom operators concerning the various logical rules, and 2 for simplifying the processing of out-of-order tuples and for considering a change in the number of points for a bonus. The amount of code we had to write compares to the amount for the custom solution. The time to implement the logic is also comparable, except the overhead to understand how Borealis works. The benefit of using Borealis is that we can use features like distribution and fault tolerance nearly for free in our system. Extending our system can be done by simply adding new operators to our network. As long as no further custom operators are required, this can be done rather easily by extending the XML file defining the query network.

As a tuple-driven SPE [4], Borealis does not use timers to close windows. A window will only be closed if a tuple of the next window arrives. As latency is important, this has become a major limitation and we had find a workaround based on punctuations and synchronization.

We can not give strong statements about performance because of

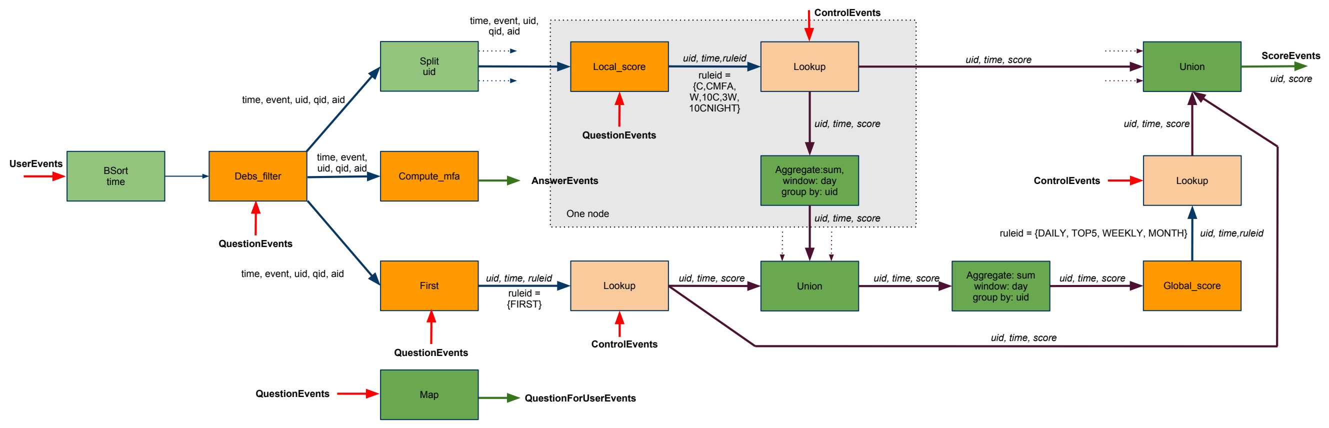

Figure 2: Borealis query network

the complexity of the system and our limited time to evaluate it. However, we observed that the performance of Borealis highly depends on the right choice of values for various tunable parameters. For example, in our measurements, we observed that a batch size which is too small results in a performance drop by more than a factor of 2.

4. DISCUSSION

In this work, we examined three different ways to implement the DEBS'11 challenge. Whilst our solutions are not highly optimized, we have learned advantages and disadvantages of each approach. To evaluate the correctness of our implementations, we created a test case generator, which implements our formalization of the challenge. All of our implementations managed to give the same results as generated. In what follows, we compare these solutions along several criteria.

- **Implementation effort:** The custom solution was quite straightforward, since it follows directly from the formal model created in advance and one does not have to fight with external APIs. Since the other two approaches are based on existing frameworks, an initial effort to setup and familiarize with the platform is to be done. Conceptually for Drools, it is reasonably easy to express the rules of the game. However, integration took time and effort since not everything behaved as described in the documentations. Configuration/tweaking required various functions which weren't well documented.
- **Performance:** Based on our limited testing, all the solutions were within the same order of magnitude in terms of overall run time. This may be partially attributed to our limited experience with each platform. One advantage is that the custom solution is a single-tier system, allowing one to optimize at will across layer boundaries as well as tweaking both data flow and memory layout. With both the rule- and stream-based approaches, however, one expresses what to compute and not how, allowing one to benefit from platform improvements. For example, with Drools, one can easily enable multi-threaded rule matching, without extensive modifications. The stream-based solution operates internally with several streams, which need to be synchronized. High latency in one operator is a bad scenario as its effects would propagate throughout the whole network.
- **Maintainability:** We were quite impressed by the model given by a rule-engine, in which state and logic are well-separated and rules are implicitly linked via events. This allows one to retain a modular structure and incrementally build an application. In principle, the same applies to a stream-engine, where one would simply re-wire the query network, as long as no custom operators are required.
- **Extensibility:** One powerful mechanism, which is available with both Borealis and Drools, is the ability to extend the provided model. This takes the form of user-defined operators in Borealis and the ability to use Java in Drools. These allow one to express logic which may otherwise be cumbersome or to tailor for performance. In both cases, we made use of this feature. A custom solution gives flexibility, but one does not benefit much from an underlying platform.
- **Architecture:** One problem we encountered with Borealis is that the operators have only local knowledge. In Drools, it was no problem to propagate the information that a day is over through the system. In the Borealis-based approach, we had to introduce punctuations and synchronization streams.

As a final note, despite having modeled the problem together, whilst we worked individually on each solution our interpretation of unclear aspects diverged. For example, we each had differing opinions on how timestamps should represent out-of-order events. This could have been alleviated with a semi-formal specification, which would leave less room for interpretation.

5. REFERENCES

[1] DEBS 2011 Grand Challenge. `http://debschallenge.event-processing.org/`.
[2] JBoss Drools - The Business Logic integration Platform. `http://www.jboss.org/drools/`.
[3] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.
[4] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. In *International Conference on Very Large Data Bases (VLDB'10)*, Singapore, September 2010.
[5] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17 – 37, 1982.
[6] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3), 2003.
[7] G. van Rossum. Python tutorial. Report CS-R9526, Apr. 1995.