Smile: Enabling Easy and Fast Development of Domain-Specific Scheduling Protocols

Christian Tilgner[1], Boris Glavic[2], Michael Böhlen[1], and Carl-Christian Kanne[3]

[1] University of Zurich
[2] University of Toronto
[3] University of Mannheim
{tilgner,boehlen}@ifi.uzh.ch, glavic@cs.toronto.edu,
kanne@informatik.uni-mannheim.de

Abstract. Modern server systems schedule large amounts of concurrent requests constrained by, e.g., correctness criteria and service-level agreements. Since standard database management systems provide only limited consistency levels, the state of the art is to develop schedulers imperatively which is time-consuming and error-prone. In this poster, we present *Smile* (declarative Scheduling MIddLEware), a tool for developing domain-specific scheduling protocols declaratively. Smile decreases the effort to implement and adapt such protocols because it abstracts from low level scheduling details allowing developers to focus on the protocol implementation. We demonstrate the advantages of our approach by implementing a domain-specific use case protocol.

1 Introduction

Modern application servers handle large numbers of concurrent requests which have to be scheduled according to, e.g., correctness criteria like classical serializability or service-level agreements (SLAs). Standard database management systems (DBMSs) offer a limited amount of fixed consistency levels, do not provide sophisticated support for SLAs and, thus, often cannot be used to satisfy domain-specific scheduling requirements. The state of the art is to develop schedulers imperatively for applications like Amazon, Ebay or Yahoo [2,5] which yields fine-tuned schedulers satisfying the application's scheduling constraints. But procedural implementations of schedulers can be complex and difficult to understand, especially if the request types and correctness criteria are less well studied than, e.g., classic serializability. Adapting schedulers to evolving requirements results in expensive and error-prone re-implementations. With our approach we address these issues by leveraging a declarative language to implement schedulers which has been shown to be beneficial in previous work [1,3].

1.1 Banking Scenario

We use the following simplified banking scenario to illustrate the shortcomings of standard DBMSs with regard to non-standard scheduling requirements. A bank institute serves normal and premium customers holding bank accounts.

A.A.A. Fernandes et al. (Eds.): BNCOD 2011, LNCS 7051, pp. 128–131, 2011.

A domain expert defines the following constraints: (C1) Account data has to be accessed under strong consistency to obviate inconsistent states and (C2) Do not schedule requests for normal customers, if there are pending requests from premium customers. How can a scheduler developer implement these constraints? Constraint C1 can be realized with standard DBMSs by applying a high isolation level, but C2 is not supported by standard DBMSs. The alternative is to develop a new scheduler from scratch which is expensive and error-prone.

2 Smile: Declarative Scheduling Middleware

Smile, our declarative scheduling middleware prototype, allows the implementation of domain-specific scheduling constraints. Executable scheduling protocols are specified with few lines of code, paving the way for sophisticated and easy-to-reason-about scheduling protocols. Our approach is based on a generic formal framework called Oshiya[1] that models the scheduling state as a set of so-called *scheduling relations*, e.g., one relation stores the schedule produced so far. Scheduling logic is encapsulated in a set of declarative queries called *scheduling queries*. To produce a schedule (sequence of scheduling relation states), Smile schedules multiple requests at the same time by repeatedly executing the scheduling queries over the scheduling relations.

This approach has several advantages: **(1)** Smile abstracts from low level scheduling details that are independent of the scheduling constraints such as parallelism in the scheduler code, queueing of incoming requests, or managing (network) connections. Developers can focus on the protocol implementation (the scheduling queries) itself, which decreases the amount of code and the effort needed to implement or adapt schedulers. We developed scheduling queries for the strong two-phase locking (SS2PL) protocol [4] as well as for a data dependent, relaxed consistency protocol. **(2)** Smile's underlying model allows to specify scheduling protocols close to their formal definition, facilitating reasoning over properties of protocol implementations such as verifying their correctness. For instance, we have proven the correctness of the scheduling queries implementing SS2PL [4]. **(3)** The separation of scheduling logic and scheduler implementation opens up interesting optimization opportunities that we plan to investigate in future work. E.g., using specialized execution engines to execute scheduling queries and controlling the trade-off between the time spent for scheduling requests and the time spent to execute them. **(4)** Scheduling sets of requests at the same time can improve the performance for large numbers of concurrent requests [3].

2.1 Oshiya Scheduling Model

In Oshiya [3], the scheduling state (requests to schedule and history information needed for scheduling decisions) is stored in three scheduling relations: *PendingRequests* ($\mathcal{R}$) buffers requests that have to be scheduled for execution. *RelevantHistory* ($\mathcal{H}$) stores prior executed requests in their execution order, modelling

[1] Oshiya refers to the passenger arrangement staff at Japanese train stations who help to fill a train by pushing people onto the train or guiding people to free railway cars.

the schedule generated so far. *Executable* ($\mathcal{E}$) buffers requests chosen for execution. Scheduling protocols are realized as three scheduling queries: $Q_{Schedule}$, $Q_{Revoked}$, $Q_{Irrelevant}$. Given previously executed requests, these queries determine in which order to execute pending requests. The scheduler state is advanced in iterative steps by applying a generic *scheduling algorithm* (shown in Fig. 2) that evaluates the scheduling queries over the current instances of the scheduling relations. The algorithm is the same for every protocol, but is parameterized by the protocol specific scheduling relations schemata and scheduling queries. Each *scheduler iteration* (while loop) performs the following steps: **(1)** Requests scheduled in the previous iteration are removed from $\mathcal{R}$. **(2)** Newly arrived client requests ($\mathcal{N}$) are added to $\mathcal{R}$. **(3)** $Q_{Revoked}$ determines transactions that have to be aborted since the requests cannot be executed due to constraint violations or blocking. **(4)** $Q_{Schedule}$ implements the scheduling protocol. It selects all requests from $\mathcal{R}$ that can be executed in this iteration without violating the protocol constraints. **(5)** Requests in $\mathcal{E}$ are executed and **(6)** added to $\mathcal{H}$. **(7)** $Q_{Irrelevant}$ identifies those requests from $\mathcal{H}$ that are irrelevant for future scheduling decisions, and is used to prune $\mathcal{H}$ so that it does not grow continuously.

2.2 Smile Architecture

The Smile prototype implements the Oshiya scheduling model outlined in the last subsection using three threads (*ClientWorker*, *Declarative Scheduler* and *Executor*), all running independently and continuously. The Smile architecture is shown in Figure 1 with arrows denoting data flow.

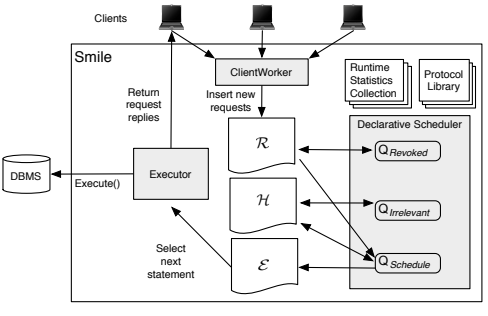

Fig. 1. Smile Architecture

$\mathcal{H} = \mathcal{E} = \mathcal{R} = \emptyset$
while true **do begin**
1 $\mathcal{R} = \mathcal{R} - \mathcal{E}$;
2 $\mathcal{R} = \mathcal{R} \cup \mathcal{N}$;
3 $\mathcal{R} = \mathcal{R} - Q_{Revoked}(\mathcal{H}, \mathcal{R})$;
4 $\mathcal{E} = Q_{Schedule}(\mathcal{H}, \mathcal{R})$;
5 $Execute(\mathcal{E})$;
6 $\mathcal{H} = \mathcal{H} \cup \mathcal{E}$;
7 $\mathcal{H} = \mathcal{H} - Q_{Irrelevant}(\mathcal{H})$;
end

Fig. 2. Smile Algorithm

ClientWorker. This thread manages client connections. The ClientWorker thread receives new requests from clients, buffers these client requests in a queue and periodically inserts them into $\mathcal{R}$ as batch job (Step 2).

Declarative Scheduler. This thread performs request scheduling by periodically executing $Q_{Revoked}$, $Q_{Schedule}$ and $Q_{Irrelevant}$ (Steps 3, 4, 7).

Executor. The Executor thread is executing the scheduled requests located in $\mathcal{E}$ against the DBMS by repeating the following steps: Retrieve the request with the smallest *ID* from $\mathcal{E}$, execute it against the back-end DBMS, return the request result to the client that has sent this request, and delete it from $\mathcal{E}$ (Step 5).

Protocol Library. Smile offers a protocol library providing the scheduler developer with pre-cooked scheduling queries (e.g., for SS2PL). These scheduling queries can be used out of the box or as a starting point to develop domain-specific protocols. We expect developers to extend this library over time with their own protocol modules.

Runtime Statistics Collection. We let Smile gather statistics about the behaviour of its operations at runtime such as the cardinalities of $\mathcal{R}$, $\mathcal{H}$ and $\mathcal{E}$ and the execution times of the scheduling queries. In future work, we plan to expose this information to the scheduler developer for the use in the scheduling queries and let her provide policies for scheduling the execution of the Smile threads.

We developed strategies deciding when to pause a thread ensuring an efficient resource usage. E.g., running the Executor while $\mathcal{E}$ is empty wastes resources.

2.3 Example: Use Case Implementation

We sketch the protocol implementation of the use case to illustrate the simplicity and conciseness of our approach. Scheduling queries are given as domain relational calculus (DRC) expressions. A simplified DRC formulation of $Q_{Schedule}$ implementing the use case constraints is:

$$Q_{Schedule} = \{S, C \mid is2PL(S, C) \wedge \\ \exists C_2(C_2 =' premium' \wedge (is2PL(\_, C_2) \Rightarrow C =' premium'))\}$$

We use a declarative implementation of SS2PL to realize constraint C1. Predicate $is2PL(S, C)$ uses $\mathcal{R}$ to determine all requests S with their customer class C that can be executed without violating the SS2PL constraints that have to hold for the generated schedule (requests in relation $\mathcal{H}$). We use S as a shorthand for the request related attributes of $is2PL$ (transaction ID etc.). Using Oshiya, we can implement scheduling constraint C2 as follows: If there exists at least one request of a premium customer ($C_2 =' premium' \wedge is2PL(\_, C_2)$), then only premium requests are selected by $Q_{Schedule}$ ($C =' premium'$).

References

1. Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J.M., Sears, R.: Boom Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In: EuroSys, pp. 223–236 (2010)
2. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., Yerneni, R.: PNUTS: Yahoo!'s Hosted Data Serving Platform. PVLDB 1(2), 1277–1288 (2008)
3. Tilgner, C.: Declarative Scheduling in Highly Scalable Systems. In: EDBT/ICDT Workshops, pp. 41:1–41:6 (2010)
4. Tilgner, C., Glavic, B., Boehlen, M., Kanne, C.-C.: Correctness Proof of the Declarative SS2PL Protocol Implementation. Technical Report IFI-2010.0008, University of Zurich, Department of Informatics (2010)
5. Vogels, W.: Data Access Patterns in The Amazon.com Technology Platform. In: VLDB, vol. 1 (2007)