View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by OAR@UM

An Architecture Supporting Compensation-Aware Monitoring*

Christian Colombo
Dept. of Computer Science
University of Malta
christian.colombo@um.edu.mt

Gordon J. Pace
Dept. of Computer Science
University of Malta
gordon.pace@um.edu.mt

Patrick Abela
Ixaris Ltd
Malta
patrick.abela@ixaris.com

ABSTRACT

To avoid large overheads induced by runtime monitoring, the use of asynchronous log-based monitoring is sometimes adopted — even though this implies that the system may proceed further despite having reached an anomalous state. Any actions performed by the system after the error occurring are undesirable, since for instance, an unchecked malicious user may perform unauthorized actions. Since stopping such actions is not feasible, in this paper we investigate the use of compensations to enable the undoing of actions, thus enriching asynchronous monitoring with the ability to restore the system to the original state in which the anomaly occurred. Furthermore, we show how allowing the monitor to adaptively synchronise and desynchronise with the system is also possible and report on the use of the approach on an industrial case study of a financial transaction system.

1. INTRODUCTION

The need for correctness of systems has driven research in different validation and verification techniques. One of the more attractive approaches is the use of monitors on systems to verify their correctness at runtime. The main advantage in the use of runtime verification over other approaches, is that it is a relatively lightweight approach and scales up to large systems — guaranteeing the observation of abnormal behaviour.

Even though monitoring of properties is usually computationally cheap when compared to the actual computation taking place, the monitors induce an additional overhead, which is not always desirable in real-time, reactive systems. In transaction processing systems, the additional overhead induced by each transaction can limit throughput and can cripple the user-experience at peak times of execution. One approach usually adopted in such circumstances, is that of evaluating the monitors asynchronously with the system, possibly on a separate address space. The overhead is reduced to the cost of logging events of the system, which will be processed by the monitors. However, by the time the monitor has identified a problem, the system may have proceeded further.

In this paper, we adopt the use of the notion of compensations from long-lived transactions in our setting to enable the undoing of system behaviour when an asynchronous monitor discovers a problem late, thus enabling the system to rollback to a sane state. We propose an architecture to enable loosely-coupled execution of monitors with the system, typically running synchronously, but allowing for desynchronisation when required and re-synchronisation when desired.

*The research work disclosed in this publication is funded by the Malta National Research and Innovation (R&I) Programme 2008 project number 052.

2. COMPENSATIONS

Two major changes occurred which rendered traditional databases inadequate in certain circumstances [4, 3]: on the one hand there was the advent of the Internet, facilitating the participation of heterogeneous systems in a single transaction, and on the other hand, transactions became longer in terms of duration (frequently, the latter being a consequence of the former). These changes meant that it was possible for a travel agency to automatically book a flight and a hotel on behalf of a customer without any human intervention — a process which may take time (mainly due to communication with third parties and payment confirmation) and which may fail. These issues rendered the traditional mechanism of resource locking for the whole duration of the transaction impractical since it may cause severe availability problems, and motivated the need for a more flexible way of handling transactions amongst heterogeneous systems while at the same time ensuring correctness. A possible solution is the use of compensations [4, 3] which are able to deal with partially committed long-lived transactions with relative ease. Taking again the example of the flight and hotel booking, if the customer payment fails, the agency might need to reverse the bookings. This can be done by first cancelling the hotel reservation followed by the flight cancellation, giving the impression that the bookings never occurred.

3. A COMPENSATION-AWARE MONITORING ARCHITECTURE

LARVA [2] is a synchronous runtime verification architecture supporting DATEs [2] as a specification language. A user wishing to monitor a system using LARVA must supply a

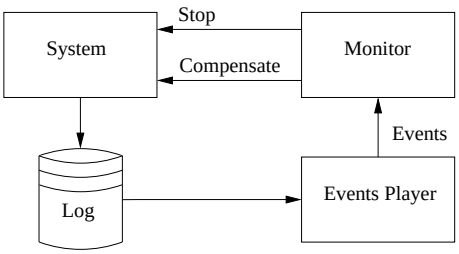

Figure 1: The asynchronous architecture with compensations cLarva.

system (a Java program) and a set of specifications in the form of a Larva script — a textual representation of DATEs. Using the Larva compiler, the specification is transformed into the equivalent monitoring code together with a number of aspects which extract events from the system. Aspects are generated in AspectJ, an aspect-oriented implementation for Java, enabling automatic code injection without directly altering the actual code of the system. When a system is monitored by Larva generated code, the system waits for the monitor before continuing further execution.

We propose an asynchronous compensation-aware monitoring architecture, cLarva, with a controlled synchronous element. In cLarva, control is continually under the jurisdiction of the system — never of the monitor. However, the system exposes two interfaces to the monitor: (i) an interface for the monitor to communicate the fact that a problem has been detected and the system should stop; and (ii) an interface for the monitor to indicate which actions should be compensated. Therefore, the actual time of stopping and how the indicated actions are compensated are left for the system to decide.

Fig. 1 shows the four components of cLarva and the communication links between them. The monitor receives system events through the events player from the log, while the system can continue unhindered. If the monitor detects a fault, it communicates with the system so that the latter stops. Depending on the actions the system carried out since the actual occurrence of the fault, the monitor indicates these actions for compensation. It is important to point out that the monitor can only compensate for actions of which it is aware — the monitor can never alert the system to compensate actions which have not been logged.

To support switching between synchrony and asynchrony, a *synchronisation manager* component is added as shown in Fig. 2. All connectors in the diagram are synchronous with the system not proceeding after relaying an event until it receives control from the manager. The following code snippet shows the logic of the synchronisation manager:

```
c = ok                    ;set default control to ok
while (c != stop)
  if (synch_mode)
    e = in_event()        ;read event from system
    c = out_event(e)      ;forward to monitor and get
                           its resulting state
    out_control(c)        ;relay control to system
  else
    par                   ;parallel execution
```

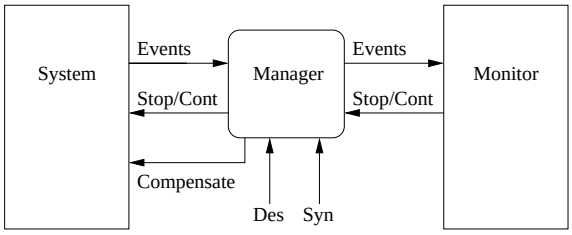

Figure 2: The asynchronous architecture with synchronisation and desynchronisation controls.

```
      e1 = in_event()     ;read from system
      addToBuffer(e1)     ;store in buffer
      out_control(c)      ;return control to system
    with
      e2 = readFromBuffer() ;read from buffer
      c = out_event(e2)   ;forward to monitor and get
                           its resulting state
end
```

The behaviour in which this architecture differs from cLarva is that it can operate in both synchronous and asynchronous modes and can switch between modes. Switching from synchronous to asynchronous is trivial. The opposite requires that the manager waits for the monitor to consume all the events in the buffer and then allowing the system to proceed further. So far this has not been implemented, but we aim to implement it in the future as an improvement on cLarva.

In real-life scenarios it is usually undesirable to stop a whole system if an error is found. However, in many cases it is not difficult to delineate parts of the system to ensure that only the relevant parts of the system are stopped. For example, consider the case where a transaction is carried out without necessary rights. In such a case, the transaction should be stopped and compensated. However, if a user has managed to illegally login and start a session, then user operations during that session should be stopped and compensated.

4. CASE STUDY

We have applied cLarva on Entropay, an online prepaid payment service offered by Ixaris Systems Ltd[1]. Entropay users deposit funds through funding instruments (such as their own personal credit card or through a bank transfer mechanism) and spend such funds through spending instruments (such as a virtual VISA card or a Plastic Mastercard). The service is used worldwide and thousands of transactions are processed on a daily basis.

We monitored four main types of properties: (i) life-cycle properties (eg. a user may only perform monetary operations if s/he has been through registration, activation and a successful login); (ii) real-time properties (eg. a user account which is inactive for more than six months should be deactivated); (iii) rights properties (eg. a user cannot login without a login right); and (iv) amount properties (eg. a user cannot transfer more than X euros per day).

The case study was successfully executed on a database of 300,000 users with around a million credit cards. A number

[1] www.ixaris.com

of issues have been detected through the monitoring system: (i) certain logs were missing; (ii) some users were found to be in a wrong state, eg. should be in a frozen state but still active; (iii) the limit of the amount of money a user can spend was in some cases exceeded. Monitoring of the logs performed asynchronously ensured the identification of issues, and through the compensation mechanism, identification of actions to be taken to rollback the system to the point where the violation occurred. At that point, one can then either notify the operator of the issue, or trigger the system's own exception handling mechanism.

5. CONCLUSIONS

In this paper, we have presented an adaptive compensation-aware monitoring architecture, and an implementation cLarva. Combined with the notion of compensations where actions of a system can be 'undone' to somewhat restore a previous state, we reduce the effect of errors detected late (due to asynchronous monitoring) by compensating for additional events which the system may have performed in the meantime. We have demonstrated the use of this approach on a financial transaction handling software. The advantage of this case study is that compensations were already a well-defined concept from the developers perspective.

This paper is an extended abstract of work which has been accepted for publication [1].

6. REFERENCES

[1] C. Colombo, G. J. Pace, and P. Abela. Compensation-aware runtime monitoring. In *Runtime Verification*, 2010. To appear in Lecture Notes in Computer Science, Springer.

[2] C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 5596 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 2008.

[3] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259. ACM, 1987.

[4] J. Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, Proceedings*, pages 144–154. VLDB Endowment, 1981.