On the Consistent Observation of Active Systems

Gianluca Moro*, Antonio Natali, Mirko Viroli
DEIS - University of Bologna, Italy
via Rasi e Spinelli 176, I-47023 Cesena (FC)
{gmoro,anatali,mviroli}@deis.unibo.it

Abstract

The consistency issues have been well studied in term of methodologies and technologies for the most popular interaction paradigm, namely the client-server. In the agent's world such a paradigm is used as much as another one, the notification-observation, in which an active system notifies its status changes to registered observer agents. In the client-server paradigm, when a client invokes a read operation of the system, the latter is always able to reply with a consistent status. Also in the notification-observation paradigm, the perception of consistent status changes of the observed system is a requirement, in order to take decision correctly. But the above-mentioned instruments used for the client-server scenario, are not suited for the notification-observation paradigm. In this paper we highlight the problem and propose an abstraction called observation interface, as an architecture allowing observer agents to perceive consistent status changes of an active system.

1. Introduction

The even widening dissemination of active systems (e.g. monitoring, workflow, active databases, active Web sources, decision support system, etc.) has generated a new tendency in the software engineering, based on the observation of the sources of events.
Observers are reactive agents notified by the system when its status changes; in other words they do not retrieve the status of the system on a polling basis. Systems to be observed are often made up of several components and their global status depends on the status of these components. The observers are not guaranteed to view a global (or partial) consistent status just by listening directly the notifications of the single sub-components.
The literature has proposed methodologies and successful technologies which allow concurrent proactive clients to update and read consistently the status of a system (e.g. DBMS Transaction Manager [4][2], CORBA Transaction Service [6], Transaction Processing Systems [1][5], Enterprise Java Beans [3]). Our claim is that the dual problem, that of systems able to notify their consistent status when necessary to reactive clients (observers), has not received the same attention.
We believe that this happens because the design of systems usually only focuses on the modeling of the entry interface (or API) used by clients (or proactive agents) to change its status (left side of Figure 1). In other words the design does not take into consideration the modeling of the dual interface (right side of Figure 1) which we call the *observation interface* (or exit interface). This interface mainly depends on constraints, validation and updating rules modeled and implemented by the system in order to preserve its consistency (function f - bottom of Figure 1). The role of the observation interface is to allow consistent observations of the global status of the system (or part of it) by notifying to observers events of a higher abstraction level.
We claim that it is not sufficient, in general, to provide reactive mechanisms to passive systems. Rather, engineering methodologies are needed to specify the set of notification events; how they map into system changes and how they are related. We believe that the idea of observation interface for an active system can represent a first step in this direction. For example it could be an add-on for a new generation of observation-compliant TP monitors.
The remainder of this short paper is organized as follows. The next section highlights the problem through a symbolic example. Section 3 proposes a framework for the design of the observation interface for active systems and in the concluding section we outline an implementation as a "proof of concept" of this framework.

2. The Issue of Consistent Observation

Typically a system is thought out and designed by considering the point of view of its use, that is by providing a set of operations to some clients in order to perform elaboration

*Contact author: Gianluca Moro, Tel +3954722221, Fax +3954722418

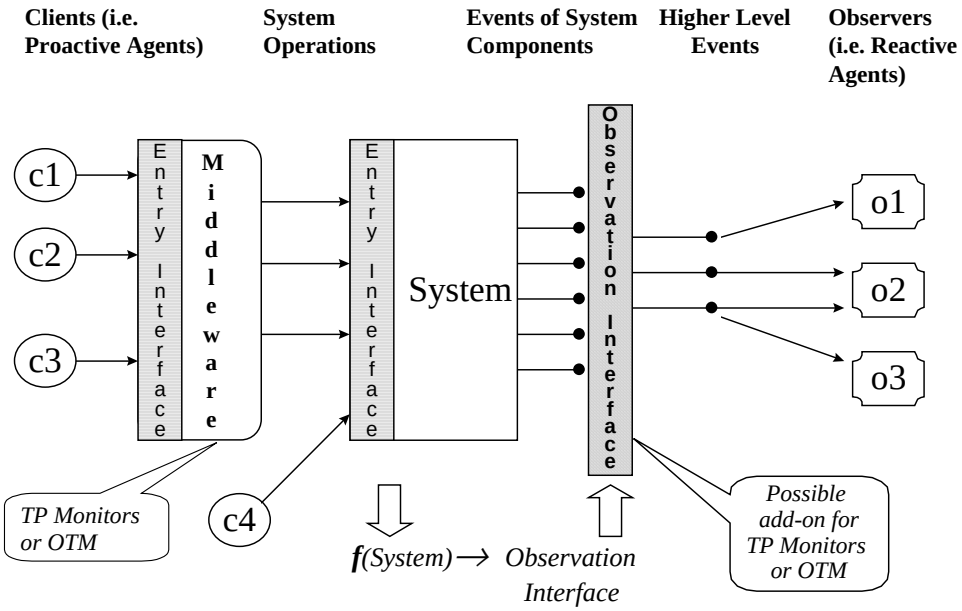

Figure 1. Entry and Observation Interface of a system

and possibly to change its status. Generally the system is designed to serve concurrent requests from proactive clients preserving its internal consistency (i.e. constraints and rules of the modeled information). This means that when clients of such a system invoke update operations, some components of the system change their status, and the system transits from an old status towards a new one. The consistency is guaranteed because the operations are executed through transactions which modify atomically the status of system components. Each time a transaction commits, the system goes from an old consistent status towards a new consistent one.

Figure 2 shows a simple example of a system, without losing generality, made up of 3 components a, b, c, representing three integers. The system transits through status $S1$, ...,$S4$ because of a transaction T_{bc} carried out by a proactive client (or agent). In the status $S2$ the transaction T_{bc} is changing the component c setting it to 5, and in the status $S3$, T_{bc} is changing the component b setting it to 0. $S2$ and $S3$ are intermediate states of the system, in which the transaction could also abort, rollbacking the system at the initial status $S1$. After the commit at time $T4$, the system reaches the consistent status $S4$ in which all components are permanently changed.

Traditionally if a proactive client asks to read the status of the system during the execution of transactions, the system replies by returning a consistent status anyway. In this case if a client invokes a status read at time $T2$, the system will return the status $S4$ or the status $S1$ depending on the isolation level chosen by the transactions [4][2].

As mentioned in the previous section, effective models and technologies allow the realization of such a system for updating and reading consistently its status independently from the number of concurrent proactive clients.

In a relevant class of current applications an active system is observed in order to take decisions or to perform some elaboration. The main difference of an active system from the previous one is that it should be able to notify its status changes to remote observers. The objective of an observer is similar to that of a client invoking a consistent status reading, the main difference is that the observer does not emit such a request on a polling-basis because it is notified by the system itself only when necessary.

Currently both areas of Object-Oriented and DBMS technology offer implementation mechanisms for the observation of their single components, generally based on the event-driven paradigm. For instance, if the components of the system of Figure 2 are tables on a relational database, they could be observed by placing a trigger on each of them ([9][10]). Each trigger notifies to the observer every change occurring due to the transaction T_{bc}. In this way the observer observes all states, included $S2$ and $S3$ which are not committed, giving raise to observations which are not consistent. In fact let us consider for example an observer interested in the sum of components a, b, c of Figure 2, considered as three distinct bank accounts, and the transaction T_{bc} as an usual bank transfer operation. Each time the observer receives a change it updates the sum. At time $T1$ the sum is 6, at time $T2$ it is 11 and at time $T3$ and $T4$ it is 6 again. In conclusion the observer at time $T2$ has ob-

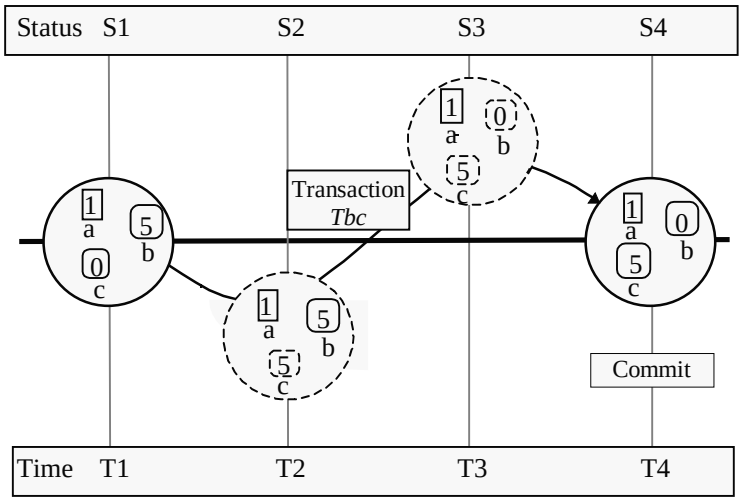

Figure 2. Status changes of a system due to a transaction

served incorrect account balances. Additionally if a transaction fails, a rollback occurs, and the observer may have performed decisions or elaborations on the basis of a status which does not exists any more. In other words the issue is analogous to that of a client which reads inconsistent states, but the problem requires different solutions. These considerations are quite general and also hold if the system components of Figure 2 are objects, simply the notification happens through the reactor-acceptor pattern [8].
To avoid incorrect observations, the active system must only manifest consistent states. The next section proposes a first solution based on what we call *observation interface*, by which observers can know what can be observed and how they can register itself for several kinds of observation.

3. The Observation Interface

The architecture we propose to manage the problem of the consistent observation of active systems has already been depicted in Figure 1. Observers do not have to directly access the reactive components of the system, but communicate with a software component, which we call the observation interface. Such component transforms the notification events that the system sends, the *system events*, into the events which will be notified to the observers, the *output events*. The former are in general of an higher abstraction level than the system events; they will be obtained by performing some computation over the data carried by the system events. Furthermore such computations will only be realized using the data values coming from the same committed transaction. The observation interface, so to speak, will take account of the problem of the notification of consistent views. In this section we will sketch a possible design for this software component. The process of transformation that the observation interface realizes is in general not straightforward and comprehends a set of subfunctionality whose implementation may depend on the application domain, on the way the system deals with transactions, on the type of information the observers may need. For these reasons we split the work of the observation interface into three, different and mostly independent subcomponents (see Figure 3):

- *Synchronizer*: which accepts the system events and decides how to group them into wholes, for successive processing
- *Evaluator*: which realizes the actual mapping from the data carried by the system events and the data that will be carried by the output events
- *Notifier*: which manages the communication with observers

In order to simplify the comprehension of the tasks these sub-components realize, we will refer to the example shown in Figure 2, that is:

- the active system may send 3 kinds of system events, namely eA, eB e eC. Each is the notification of a change on one of the components of the system, which are a, b or c, and carries its new status, represented as an integer value. Furthermore the observation interface has also to accept two more events: eTC which is fired when a transaction commits, or eTR when a transaction rollbacks, both carrying the transaction identifier TrID;

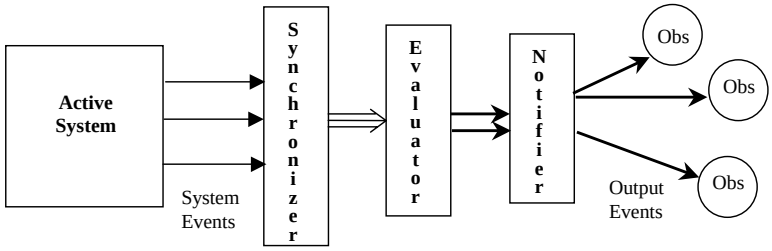

Figure 3. Observation Interface details

- the observation interface exports three kinds of output event: $o1$, $o2$ e $o3$. $o1$ will carry an integer value which is the sum of the values carried by eB and eC, $o2$ the sum of those of eA and eB, and in the case of $o3$ just the data value carried by eA. We will say that $o1 = eB + eC$, $o2 = eA + eB$, $o3 = eA$;
- three observers, $Ob1$, $Ob2$ and $Ob3$ are registered for respectively $o1$, $o2$ and $o3$;

Synchronizer. The strategy of grouping system events into wholes is stored in the Synchronizer. The Synchronizer is in fact the only component of the observation interface which has to know the details of the System, and in particular about the way information is packed into events. It accepts asynchronous system events and keeps track of them. Then by applying some synchronization policy, that can be local or based on the information the events carry, it can decide to put together some of them and send the whole to the following processing. The typical implementation is that in which events belonging to the same transaction are collected (using their TrID), and fired when the transaction commits (events eTC) or discarded if rollbacks occurs (event eTR). In our use case an event of type eC carrying the value 5 arrives at time $T2$, and an event of type eB with value 0 arrives at time $T3$. When at time $T4$ the transaction commits the two couples, $(eC, 5)$ and $(eB, 0)$, are sent for further processing.

Evaluator. An observation interface typically exports a finite set of output events. The task of the Evaluator is to accept the group of events it receives from the Synchronizer and to compute the output data values that will be carried by the output events. Such events will be used by the observers to take their decisions and to perform some new computation, in a consistent way. This is the core of the mapping process, and can be typically realized in a functional way, storing an internal status when needed. In general the data values which arrive from the Synchronizer need an adjustment before the computation can be executed. For example there can be more values for a single input (caused by the arriving of two events of the same kind of system event), or not all the inputs may have values to be computed. Typically the adjustment will discard older values in the first case and use a cached copy for the second. In our example the Evaluator receives $((eC, 5), (eB, 0))$, adjusts it by using the cached value of eA (suppose it is 1) obtaining $((eC, 5), (eB, 0), (eA^*, 1))$, and performs the computation which returns $((o1, 5), (o2, 1), (o3^*, 1))$. We used the star symbol over eA and $o3$ to highlight the fact that their values are not new, but obtained using a cached copy. Notice that we admit $o2$ to be computed using a new value (that of eB) and a cached value (eA^*), and that we do not consider $o2$ a cached copy.

Notifier. This is responsible for handling the communications with the observers. First of all it accepts observers registration for some of the output events it provides and keeps track of them using a queue structure. Then it receives the data computed by the Evaluator and prepares the correspondent output events. In our use case it will notify the observer $Ob1$ with the value 5 and the Observer $Ob2$ with the value 2. Observer $Ob3$ is not notified because $o3$ is star-labelled. Notice in fact, that the component a of the system is not changed by the underlying transaction.

4. Conclusions

Typically the observers listen to the notification events sent by an active system in a direct way but this may cause the observers to perceive status which are not consistent. The situation becomes even more problematic in the case of concurrent transactions, and transactions which can rollback. Notice that in the latter case events would be fired anyway and it is not possible in general to obtain a rollback of the actions the observers has performed due to the events that have already been generated. The observation interface introduced in this paper helps to overcome this problem, notifying only consistent states of the system. An implementation of the framework we presented has been realized as a "proof of concept" over an Oracle Database (version 7.3) [7], which is available at the following URL: http://elena.ingce.unibo.it/observation/. Obviously additional work has to be done, both from a theo-

retical and from a pragmatical point of view, to better understand the impact of the consistent observation issue on the engineering of active systems.

References

[1] Philip A. Bernstein, Transaction Processing Monitors, *Communication of ACM*, november 1990/Vol.33,No.11, pag. 76-86

[2] A. Elmasri (Ed.): Database Transaction Models for Advanced applications, *Morgan Kaufmann Publishers*, Inc, 1992

[3] Enetrprise Java Beans Specification, 1999, *Sun Mycrosystems*, http://www.javasoft.com/products/ejb/docs.html

[4] J. Gray, A. Reuter: Transaction Processing: Concepts and Techniques. *Morgan Kaufmann* 1993, ISBN 1-55860-190-2

[5] Sherri Kennamer, Microsoftcorn: A High-Scale Data Management and Transaction Processing Solution, *SIGMOD'98* Seattle. WA, USA, 1998 ACM pag. 539-540

[6] R. Orfali, D. Harkey, Client/Server Programming with Java and CORBA, *Wiley Computer Publishing*, USA, ISBN 0-471-24578

[7] *Oracle Corporation*, PL/SQL User's Guide and Reference Release 2.3, Part No. A32542-1, chapter 8, February 1996

[8] Schmidt, D. Reactor : An object behavioral pattern for concurrent event demultiplexing and event handlaer dispatching. In J.O. Coplien and D.C. Schmidt, Eds., Pattern Languages of Program Design. *Addison-Wesley*, Reading, Mass., 1995

[9] Widom, J., Ceri, S., Active Database Systems. *Morgan Kaufmann*, San Matteo, Calif., 1996

[10] Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R. T., Subrahmanian, V. S., Zicari, R., Introduction to Advanced Database Systems, *Morgan Kaufmann*, San Matteo, Calif. 1997