Efficient Trigger Management in Multiagent Systems

Kornelije Rabuzin, Mirko Maleković
Faculty of Organization and Informatics
University of Zagreb
Pavlinska 2, 42000 Varaždin, Croatia
{kornelije.rabuzin, mirko.malekovic}@foi.hr

Abstract. *The active database theory is well developed; many different types of events are supported, static analysis mechanisms are developed, etc. One of the problems regarding active rules management is a number of rules defined; it is not so easy (for a user and for a system) to manage a large number of rules. Although some systems allow similar rules grouping, the problem is still significant. During a project we had the same problem occurred i.e. a number of time triggers that were used was too big. This paper describes how we reduced a number of time triggers that were used in a multiagent system by introducing a meta-trigger.*

Keywords. Trigger management, active databases, reactive agents

1 Introduction

Agents and multiagent systems were (are) used in many different areas for solving many different problems. Many papers were published and interesting research conducted and reported in published publications. Conflict resolution strategies are just one of the problems still present in multiagent theory that gain much attention. The problem which we had and tried to solve referred to efficient resource allocation in a multiagent system; how to efficiently allocate some set of scarce resources and avoid conflicts was the main issue. The problem was described in [7]. A database we had contained all relevant information about resources that were allocated and many deliberative agents had to allocate these resources efficiently. Although deliberative agents were able to allocate resources and solve conflicts that occurred during the allocation, the solution was far from optimal. Agents often run into deadlocks while allocating resources (Figure 1), and negotiation took too much time [8]. A new, fast and reliable solution had to be found.

In order to resolve conflict situations one has to understand what a conflict is, what could cause a conflict, and how conflicts could be resolved [8]. In [8] several different conflict definitions were presented, conflict causes were identified, different types of conflicts were described, and conflict resolution strategies presented as well. We can say that two basic strategies for handling conflict situations could be applied; avoid or resolve ([12]). Conflicts could be resolved in two ways: cooperative and non-cooperative [13], [14]. Non-cooperative conflict resolution methods are in general dealing with worst-case scenarios [14], while cooperative algorithms involve information exchange between the agents. Cooperative approaches can be further broken down into centralized and decentralized methods [6]. The first approach subsumes that there is a central component that resolves conflicts, while the second approach is based on coordination protocols [8]. Although the solution can be optimal, gathering all information at a central location might be a challenging task in practice if the system is large [6]. According to [4] there is no universal way to resolve conflict situations; that's why they tried to define a framework to resolve conflicts in a way that different techniques for resolving conflicts were integrated, and the best one was selected according to the conflict that occurred. Although the proposed solution was good, time was spent to determine which strategy to use.

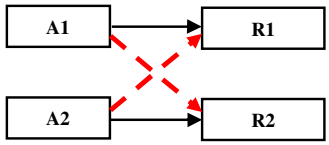

Figure 1. A physical conflict

All mentioned approaches are trying to solve conflict situations, but it seems much better to avoid them (if possible). In order to avoid conflicts, based on conducted research regarding conflicts, we didn't want to define some new architecture, but we decided to use existing technologies and rules of thumb. During the years researchers have come to the conclusion that reactivity is also a very important characteristic that an intelligent agent should possess [7]. Reactivity is suitable for dynamically changing

environments performing an immediate response to some changes which have been recognized and perceived [8].

Since a database with needed data (information) existed, in order to resolve conflicts it was decided to build a fast centralized solution. For that purpose a reactive database agent was introduced and implemented within the database by means of stored procedures and event-condition-action rules (in order to fulfill the speed requirement). Since triggers were placed on a server, and their execution time can be neglected, the solution possessed very good computational performances. In order to build such an agent the connection between active databases and reactive agents was explored in a detail, and results presented in [10].

Active databases and reactive agents were developed independently and only a few articles were written on the subject. What is common to both fields is that both types of systems can perceive the environment and react to recognized events. Further on, events are very important component for both systems, although authors in [1] consider that active systems mainly use simple events. Actions that are executed in agent systems may be evaluated; in active databases this possibility is not directly supported [3]. The conditions are rarely mentioned in the context of reactive agents, but are intrinsic to active databases. In agent systems events are simpler, but actions may include complex plans and reasoning that is beyond predefined scripts that are used in active databases [10]. Finally, triggers are more complex today because one can use many programming languages to specify actions that are going to be executed. Active databases support transactions, while agents do not. It is very hard to ensure atomicity in agent systems; the order of actions is crucial if an agent tries to execute some set of actions successfully. The problem that is still present in agent systems is the question of rollback; how to ensure that, after some actions were already executed, the system restores to a previous state and that effects of performed actions are cancelled (ignored). We investigated and established a connection between active databases, reactive agents and business rules and published the results in our previous paper.

A database agent that was developed and added into the multiagent system allocated resources within transactions; atomicity ensured that either all needed resources were efficiently allocated, or none. In that way deadlocks were avoided and some set of resources was allocated efficiently (that is not so easy to ensure in decentralized environment).

The problem with the system was how to ensure that already taken resources were de-allocated. For example, an agent could have taken several resources and then decided not to return them. In that case other agents couldn't have allocated resources because they hadn't been returned yet. So due to network problems, forgetfulness, etc. there was a possibility that resources weren't returned in (on) time. So a control mechanism had to be created in order to ensure that resources were returned and available for other agents; in that way physical conflicts were in fact avoided (it is much better to avoid conflicts situations than to solve them).

A control mechanism was built that was sending messages to agents and for that purpose time triggers were used. Each time a resource was taken a new time trigger was created in order to remind an agent (if a resource hadn't been de-allocated) that the resource was still in use. The number of triggers was equal to the number of taken resources; the more resources were taken, the more triggers were defined. In order to avoid the vast number of defined time triggers, their number had to be reduced.

The rest of the paper will explain how this was done. In the first section we will briefly explain some basic terms regarding active databases. Then the problem and the solution models are described, and finally the conclusion is presented.

2 Active Databases

An Active DataBase Management System (ADBMS) is a conventional database system capable of reacting to some events of interest that can occur within the database, or outside it. Usually Event-Condition-Action (ECA) rules are used to describe automatic behavior; when certain events occur (ON EVENT), and some conditions are fulfilled (IF CONDITION), then some actions are going to be executed automatically (THEN ACTION) as reactions to these recognized events. ECA rules are mostly implemented using triggers, but some systems offer some other mechanisms in order to implement active rules.

Each active database management system is based upon a passive, conventional database management system. In order to support active functionality each passive database management system has to be extended in a way that different kinds of events can be detected, transactions can be managed because of different rule execution models, etc. That can be done in three different ways: integrated, layered or application oriented approach ([2], [5]). Several approaches for active databases performance measurement were introduced as well ([5], [9]).

Events are very important components of active rules because they define when some conditions are going to be evaluated and, of course, actions executed. The SQL standard defines only three basic types of events that can be used in trigger specification: INSERT, UPDATE, and DELETE. But for solving real problems many different types of events were introduced during the years as well [5], [9].

Events can be divided into simple and complex events [5]. Complex events are mostly based on

simple ones, and simple events can be divided as follows:

1. Basic database operations: INSERT, UPDATE, and/or DELETE,
2. Time events:

a) absolute - certain point of time,

b) periodic - every day, month, etc., and

c) relative - for example, 30 minutes after something else happened,

3. Method events: method invocation (in object-oriented DBMS),
4. Transaction events: for example BEGIN or COMMIT, and
5. Abstract events: some user defined events.

Complex events could be built by combining several simple events, or by using several existing constructs i.e. negation, sequence, repeat, etc. For example, if we had simple events E1 and E2, then E1∧E2 or E1∨E2 would represent a complex event.

Negation represents an absence of some event within some time interval, *sequence* represents that several events occurred in a specified sequence and *repeat* means that some event (simple or complex) repeated several times within some time interval. More on composite events can be found in [5] and [11].

There are also some other important issues regarding active databases that will be briefly mentioned. Each ADBMS has a language that is used for trigger specification (definition), and has an execution model that determines how the rules are going to be executed [5].

When an event occurs the condition is evaluated and then some actions are executed, provided that condition evaluation was successful. Occasionally it makes sense to postpone condition evaluation or action execution for some time, and perform them later on. That is why several different execution models exist (immediate, deferred and decoupled).

Sometimes active databases do not exhibit desired behavior; rules could trigger one another, inconsistent rules may exist, and rule execution process may not terminate. Static analysis is used to answer all mentioned questions and to ensure more reliability during the rule execution process. Several different approaches were introduced to perform static analysis as can be found in [5] and [9].

There are also some other important issues regarding active databases, but since they are not important for this paper, will not be covered either. We refer to [5], [9] and [10].

3 The problem and the solution model

As we already described, conflict situations needed to be avoided in an efficient manner. Since optimal behavior is not so easy to achieve in decentralized environment, a centralized solution was built; a reactive agent that was built by means of active rules and stored procedures was used. The connection between reactive agents and active databases was described in [10].

The thing that was crucial for our model was how to build and ensure a control mechanism that would enable and ensure in time resource de-allocation in a multiagent system. If resources were returned in time, other agents were able to allocate them. If resources were not returned, other agents waited and that was not acceptable in a real time system we had. In order to ensure resource de-allocation, the set of active rules was defined and absolute time triggers were added into PostgreSQL DBMS, in order to remind agents that resources were still not de-allocated.

A table with resources existed. Each time a resource was allocated, a new row was added into the *resource_use* table; it was written which agent took which resource and when. Each time a new row was added into the *resource_use* table, a resource use time was estimated (based on previous information) i.e. a resource would be used for some time interval *t*. So, each allocated resource was used for some time *t*. Each time a resource was allocated (at time *T*), estimated time *t* was added and a time trigger was created to remind agent at *T1* (*T1=T + t*) that certain resource was not de-allocated yet. If a resource was de-allocated, appropriate time trigger was deleted and a message wasn't sent.

On T1

Send message to agent A1 to check resource R1

Since we had many resources allocated in each point of time, many similar triggers existed in a point of time.

On T2

Send message to agent A2 to check resource R2

…

On Tn

Send message to agent An to check resource Rn

In that way the resource status was checked, and agent was reminded to de-allocate resource(s) if they hadn't been returned. Each time some resource was returned, appropriate time trigger was deleted. Similarly, each time a resource was taken, a trigger was added.

As one can see the condition part of defined ECA rules (triggers) is missing. Conditions were not needed; the fact that trigger existed was enough because it meant that some resource was not de-allocated (otherwise the trigger wouldn't have existed). So basically if some trigger was defined than the resource hadn't been returned.

The problem with active databases is efficient management of triggers; the more triggers we have, the more difficult the management is. Although time triggers were used for notification purposes and were not dangerous in a sense that rule execution process wouldn't terminate, we decided to reduce the number of triggers and to define one meta-trigger that would perceive relevant parameters dynamically and replace

many similar triggers. As one can see, all triggers look the same; this similarity enabled us to build a meta-trigger that was active all the time, and relevant parameters were passed to this meta-trigger whenever there was a need to do so.

We used a possibility of passing the relevant parameters from the event part to the action part of the trigger defined.

ON [(T1, A1, R1), ..., (Tn, An, Rn)]
THEN send_message [(X, Y)]

The meaning of *(T1, A1, R1)* is: on *T1* send a message to an agent *A1* regarding a resource *R1*. Relevant parameters (*A1* and *R1*) were passed to the *send_message(X, Y).*; basically, a message was sent to the agent A1 to check the status of the resource R1.

In that way we had just one defined trigger that was active all the time, with many dynamically added parameters. As one can see the trigger management was easier; instead to look at the definitions of many different triggers defined, we had one file that contained all relevant parameters. The solution was implemented on a UNIX system using the CRON process; the already mentioned file that contained parameters was sent to the CRON system. For those that are not familiar with the CRON system, it operates pretty much the same as Windows Scheduler; at some point of time it performs some specified actions. On the other hand, that was a natural solution because it enabled us to directly implement our solution model.

Each time some new resource was allocated, a new row was added into the file that was used as an input for the CRON system (the following script).

```
CREATE OR REPLACE FUNCTION ioperl() RETURNS trigger AS $$
    if ($_TD->{new}{service}) {
    $query = "select (now() + avg(returned - taken)) as vrijeme from res_usage where resource ='" . $_TD->{new}{resource} . "' and service = '" . $_TD->{new}{service} . "' group by resource, service limit 1;";
    } else {
    $query = "select (now() + avg(returned - taken)) as vrijeme from res_usage where resource ='". $_TD->{new}{resource} . "' group by resource limit 1;";
        }
      $res = spi_exec_query($query);
      $time = $res->{rows}[0]->{vrijeme};
        # determine date parameters
    $godina = substr($time,0,4);
    $mjesec = substr($time,5,2);
    $dan = substr($time,8,2);
    $sat = substr($time,11,2);
    $minuta = substr($time,14,2);
    $sekunda = substr($time,17,2);
        # determine which agent took which resource
    $agent = $_TD->{new}{agent};
    $resurs = $_TD->{new}{resource};
    $servis = $_TD->{new}{service};
    $obavijest= "Agent $agent nije vratio resurs $resurs! ";
        # open and write to file...
    open(FH, ">>/home/stuff/krabuzin/perl/trigger");
    print FH "$minuta $sat $dan $mjesec * echo \" $obavijest \" \n";
    close FH;
    system ('crontab /home/stuff/krabuzin/perl/trigger');
    return;
     $$ LANGUAGE plperlu;
    CREATE TRIGGER perl_AI AFTER INSERT ON res_usage
    FOR EACH ROW EXECUTE PROCEDURE ioperl();
```

Each time a resource was de-allocated, an appropriate line was deleted from the file. So the file contained the same number of lines as the number of resources that were allocated. An example row in the file was:

*40 16 20 10 * echo " Agent A1 didn't return resource N2! "*

The meaning of the line is: on 20th of October (last year) at 16.40 a message was sent that a resource N2 was not de-allocated and was still in use by agent A1. After the message was sent several scenarios were possible; the agent could have de-allocated the resource, or for example, send a message that it was still in use.

4 Conclusion

Since many resources were allocated in the same point of time, the number of defined triggers was equal to the number of resources that were in use. In order to ensure easier trigger management, we decided to reduce the number of triggers and to introduce a time meta-trigger.

Relevant parameters were added (and removed) dynamically into just one file, and the number of triggers defined was reduced. Instead to look at many different trigger definitions, all relevant parameters were visible in just one file.

The proposed control mechanism operated successfully for a small number of resources. What has to be done is to test the proposed model in a rela-time environment including many agents and resources.

References

[1] Bailey J, Georgeff M, Kemp D, Kinny D, Ramamohanarao K: **Active databases and agent systems – a comparison**, Proceedings of the second international workshop on rules in database systems, Lecture notes in computer science 985, Athens, Greece, 1995, pp 342-356.

[2] Chakravarthy S: **Architectures and monitoring techniques for active databases**: An evaluation, Data & Knowledge Engineering, vol. 16, no. 1, 1995, pp. 1-26.

[3] Ferber J: **Multiagenten-Systeme**, Eine Einführung in die Verteilte Künstliche Intelligenz", Addison-Wesley, 2001.

[4] Liu T H, Goel A, Martin C E, Barber K S: **Classification and representation of conflict in multi-agent systems**, Technical Report TR98-UT-LIPS-AGENTS-01, The University of Texas at Austin, 1998.

[5] Paton N W: **Active rules in database systems**, Springer, New York, 1998.

[6] Purwina O, D'Andreab R, Leec J W: **Theory and implementation of path planning by negotiation for decentralized agents**, Robotics and Autonomous Systems, vol. 56, no. 5, 2008, pp. 422-436.

[7] Rabuzin K, Maleković M, Bača M: **A Combination of Reactive and Deliberative agents in Hospital Logistics**, The Proceedings of 17th International Conference on Information and Intelligent Systems, Varaždin, Croatia, 2006, pp. 63-70.

[8] Rabuzin K, Maleković M, Čubrilo M: **Resolving physical conflicts in multiagent systems**, The Proceedings of 3rd International Multi-Conference on Computing in the Global Information Technology ICCGI, IEEE Computer Society Press, Greece, IN PRESS.

[9] Rabuzin K, Maleković M, Lovrenčić A: **Extending Trigger-By-Example Approach to Support Time Events**, The proceedings of 11th International Conference on Intelligent Engineering Systems INES 2007, Budapest, Hungary, 2007, pp. 313-316.

[10] Rabuzin K, Maleković M, Ribarić S: **Implementing reactive agents in active databases**, ICITA 2008, Proceedings of 5th International ICITA conference, Cairns, Australia, 2008, IN PRESS.

[11] Tan C W, Goh A: **Composite event support in an active database**, Computers & Industrial Engineering, vol. 37, no. 4, 1999, pp. 731-744.

[12] Tessier C, Chaudron L, Müller H: **Conflicting Agents**, Kluwer Academic Publishers, Boston, 2001.

[13] Tomlin C, Pappas G, Košecka J, Lygeros J, Sastry S: **A next generation architecture for air traffic management**, available at `http://citeseer.ist.psu.edu/cache/papers/cs/3825/http:zSzzSzrobotics.eecs.berkeley.eduzSz~clairetzSzatms_cdc97.pdf/tomlin97next.pdf`, 1997, Accessed: 3rd November 2006.

[14] Tomlin C, Pappas G, Sastry S: **Conflict resolution for air traffic management**: A study in multi-agent hybrid systems, IEEE Transactions on Automatic Control, vol. 43, no. 4, 1998, pp. 509-521.